Incremental Compilation of SCEst

Milad Rahimi-Barfeh

Bachelor Thesis
September 2017

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Steven Smyth
M.Sc. Alexander Schulz-Rosengarten

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Reactive systems need determinate concurrency which is provided by synchronous languages such as *Esterel*. Their downside is a heavy restriction on programs that are accepted. By ordering statements sequentially, a programmer who is used to imperative programming languages gives information on how he wants them to be executed. This extra information is used by the Sequentially Constructive (SC) Model of Computation (MoC), which is a conservative extension of the classical synchronous MoC and Esterel, to allow more programs. The Sequentially Constructive Language (SCL) implements the SC MoC and it is possible to transform Esterel to SCL. Furthermore, the SC MoC is used by Sequentially Constructive Esterel (SCEst), which is an extension of Esterel and is grounded in SCL, which reflects in a minimal set of statements to which all complex Esterel/SCEst statements can be transformed to. These transformation rules are already defined and implemented in a monolithic transformation from Esterel to SCL following a structural approach.

In this thesis the defined transformation rules are used for an incremental compilation of SCEst programs. This allows the user to focus on the transformation of smaller parts of the program, which is more comprehensible than a monolithic compilation. For this task a meta-model which includes SCEst and SCL statements is used. The new transformations and SCEst are integrated into the KIELER framework, which allows a user to translate a SCEst program to an SCL program while being able to inspect the intermediate models of each compilation step, and subsequently SCL to target code.

Contents

List of Acronyms

ANTLR	Another Tool For Language Recognition
BLIF	Berkeley Logic Interchange Format
CEC	Columbia Esterel Compiler
DSL	Domain Specific Language
ebt	emit-before-test
EMF	Eclipse Modeling Framework
EBNF	Extended Backus-Naur Form
ELK	Eclipse Layout Kernel
GRC	graph code
IDE	Integrated Development Environment
INRIA	French Institute for Research in Computer Science and Automation
iur	initialize-update-read
KiCo	KIELER Compiler
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KLighD	KIELER Lightweight Diagrams
M2M	Model-to-Model
ENSMP	École nationale supérieure des mines de Paris
MoC	Model of Computation
PDG	Program Dependence Graph
SC	Sequentially Constructive
SCCharts	Sequentially Constructive Statecharts
SCEst	Sequentially Constructive Esterel
SCG	Sequentially Constructive Graph
SCL	Sequentially Constructive Language
SLIC	Single-Pass Language-Driven Incremental Compilation

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Nowadays embedded, reactive systems can be found almost everywhere, e.g., a smart phone, a digital watch, a car, or an airplane. Their goal is to establish a connection to the physical environment and the possibility to react to it. While dealing with the environment over sensors, they need to compute their responses and therefore it is frequently advantageous to run concurrent processes/programs, e.g., to effectively use the time that would otherwise be spent on waiting for input or output operations. There are embedded, reactive systems for which a slight deviation from their intended behavior can be tolerated, for example, when a camera should be activated by movement to capture a picture of a wild animal. Furthermore, there are so called *safety-critical* systems [MP95], which must guarantee determinism, for example, in cars or airplanes, to be predictable. But along with concurrency comes the problem of non-determinism; however, for a specific input the system must react the same way every time, or as formulated by Gérard Berry, *a system is said to be deterministic if the same sequence of inputs always produces the same sequence of outputs* [Ber00]. While normal programming languages like C do not offer that kind of guarantee by default (for example, with their use of threads and the produce of *race conditions* [Lee06]), synchronous languages like Esterel do. They divide the execution into discrete ticks. Each tick starts with an input (e.g., read over sensors), followed by a computation (e.g., calling a *tick function*), and ends with an output reaction over actuators. The downside of synchronous languages is a heavy restriction on programs that can be excepted, which comes from blocking concurrent *and* non-concurrent variable accesses as part of their semantic foundation instead of considering the sequential order in non-concurrent executions. Therefore, a better combination of expressiveness and determinism is needed, which allows for a wider range of programs to be excepted, while still guaranteeing predictability. This led to the definition of sequentially constructive languages [HMA+13], which make use of the sequential ordering of a program.

In the following, the programming languages Esterel, including a definition of the *Berry Constructiveness*, Sequentially Constructive Esterel and Sequentially Constructive Language, as well as the graphical representation, the SCG, of the latter are described. Additionally, the respective MoC is explained. Further, an overview of the previous approach of structural compilation of Esterel in KIELER by Rathlev [Rat15] is given and the SLIC approach is explained. Lastly, the problem statement is given as well as an outline for this thesis.

1.1 Esterel

A Model of Computation (MoC) defines which operations are valid in a computation [BLL+05].

The classical *synchronous* MoC divides execution into discrete ticks, which are an abstraction of physical time. The ticks define the ordering of computations. While each tick holds a *macro step*, the whole execution in it, this macro step consists of a finite amount of *micro steps* [BCE+03]. A main part of synchronous languages are *signals*, which are either *present* or *absent* in each tick, the latter by default and the former only if they are emitted. This is defined under the *signal coherence law* [Tar05].

Also, a finite reaction is executed in an instant, which means an execution takes conceptually no time and therefore the result is instantly available. While this is a valid abstraction, a computation in zero time is impossible to achieve. It suffices that the results are ready *before the next execution* begins [Ber91].

G. Berry et al. developed the synchronous programming language Esterel. It is *stable and in constant evolution* [Ber00], an imperative language, and based on the synchronous MoC [Ber00]. It is intended to be used for *programming control-dominated software or hardware reactive systems* [Ber00]. The latest versions are Esterel v5 and Esterel v7. Since Esterel v7 extends Esterel v5 by including a `weak suspend` statement that is complicated to transform, which will be explained in a moment, the focus of this thesis lies on the transformation of the Esterel v5 statements.

Pure and *valued signals* are primarily used for communication in Esterel. While pure signals just have one state, which can be absent or present, a valued signal also has a persistent value across ticks. In addition, a combine function can be defined for a valued signal, which makes it possible to emit the signal multiple times with different values during one tick by using the combine function to fuse the different values. An input signal is set by the environment, while an output signal is absent by default at the beginning of every tick.

To assume signal coherence, Esterel follows the emit-before-test (ebt) discipline, which means that the first emission of a signal has to be done before any present test of the signal. Further emissions can be scheduled arbitrarily since they do not change the presence state of the signal. Furthermore, a variable can be used in the standard way, which means its value can be sequentially overwritten multiple times in one tick. However, to guarantee determinism there are two ways a variable can be accessed by concurrent threads. In the first case, all concurrent threads have read-only access. In the second case, only one alone can read and write the variable.

The Esterel language can be divided into *kernel statements* and *derived statements*. The following variables are used to display the Esterel statements: **s** stands for a signal, while **p** stands for a single or compound statement. Esterel's *kernel statements* are [Ber93]:

signal *s* **in** *p* **end**	A local declaration of signal *s*, which covers possible previous signal declarations with the same name.
emit *s*	Set the signal *s* to present in the current tick.
p; *q*	The sequence operator. Execute *q* after *p* is finished.
p || *q*	The parallel operator. Execute *p* and *q* in parallel and finish the parallel statement after *p* and *q* have terminated.
present *s* **then** *p* **else** *q* **end**	Execute *p* when *s* is present in this tick; otherwise, execute *q*.
loop *p* **end**	*p* is executed infinitely often.
suspend *p* **when** *s*	When *s* is present in a tick, suspend the execution of *p* for this tick.
exit *T*	Exit the surrounding **trap** statement by jumping to the end of it. In case of nested traps, the outermost trap has priority. The execution of a concurrent thread stops its execution not until the next tick boundary.
trap *T* **in** *p* **end**	*p* is executed normally or until an **exit** statement invokes *T*.
nothing	An empty statement which does nothing.
pause	For the current tick the execution stops.

All *derived statements* of Esterel v5 can be transformed to *kernel statements* [Ber93]; therefore, they are syntactic sugar, which make it possible to write a more compact program that is also

```
halt
```

```
l1:
pause;
goto l1;
```

Figure 1.1. Transformation of the `halt` statement of Esterel to kernel statements.

more comprehensible. Examples are the `abort`, `await` or `halt` statement. The `halt` statement pauses indefinitely and thus stops the execution. It is shown in Figure 1.1. `halt` is transformed to a label, followed by a `pause` statement and lastly a `goto` statement, which activates a jump back to the label.

Figure 1.2 shows the `ABRO` example as an Esterel program with derived statements (Figure 1.2a) and with kernel statements (Figure 1.2b). The advantage of the former is its comprehensibility compared to the latter. When input signal `A` and `B` have been present at least once, output signal `O` is emitted. The input signal `R` stands for reset and therefore, in case of being present in a tick, `O` is not emitted despite of the states of the signals `A` and `B` and the whole behavior is reset because of the strong preemption.

Esterel can be compiled to a host language, for example Java or C. A problem is in guaranteeing deterministic behavior despite of concurrency. Therefore, the interaction between threads is handled statically at compile time. By using a host language, it is possible to use the host languages features, for example when compiled to Java, the data types of Java can be used in addition to the few types of Esterel.

In Esterel v7 the new statement `weak suspend` is added. When activated, the current state of the execution is saved, so the execution in the next tick can start at this point, but the current tick is completely executed as if the `weak suspend` statement was not activated. Since tick boundaries depend on signals, they can not be determined statically in conjunction with the new `weak suspend` and therefore it is quite obvious that `weak suspend` can not be transformed to the already existing kernel statements.

Berry Constructiveness

The *constructive coherence law* [Ber02] reads as follows:

- ▷ A signal is declared present if and only if it must be emitted.
- ▷ A signal is declared absent if and only if it cannot be emitted.

They define the *constructive semantics* of Esterel. The constructive semantics also consider an *unknown* state beside absent and present. It is sufficient to determine all states of the output signals to be either present or absent to consider a program *B-constructive*, or in short *constructive*, by realizing which signals *must* or *cannot* be present.

Figure 1.3 shows two examples for constructiveness. The program in Figure 1.3a is excepted as it is a constructive Esterel program since the output signal `B` must be emitted because of line 3. The program in Figure 1.3b is rejected since the output signal `B` can be emitted but does not have to be emitted as no speculation of which branch has to be taken is allowed.

```
module ABRO_derivedS:
input A,B,R;
output O;
loop
  [
    await A
    ||
    await B
  ];
  emit O
each R
end module
```

(a) ABRO with derived statements.

```
module ABRO_t1:
input A,B,R;
output O;
loop
  trap T1 in
    suspend
      [
        trap T3 in
          loop
              pause;
            present A then
                exit T3
              end present
          end loop
            end trap
        ||
        trap T4 in
          loop
            pause;
            present B then
              exit T4
            end present
          end loop
        end trap
      ];
      emit O;
      halt
    when R;
    exit T1;
    ||
    trap T2 in
      loop
        pause;
        present R then
          exit T2;
        end present
      end loop
    end trap;
    exit T1;
  end trap
end loop
end module
```

(b) ABRO in kernel statements.

Figure 1.2. ABRO example with derived statements and only with kernel statements.

```
module constructiveness_1:
output B;
emit B;
present B then
  emit B
else
  emit B
end present
end module
```

(a) A constructive Esterel program.

```
module constructiveness_2:
output B;
present B then
  emit B
else
  emit B
end present
end module
```

(b) An Esterel program which is not constructive.

Figure 1.3. An example of constructive and not constructive Esterel programs.

1.2 Sequentially Constructive Languages

The Sequentially Constructive (SC) Model of Computation (MoC) is a *conservative extension in that programs considered constructive in the common synchronous MoC are also SC and retain the same semantics* [HMA+13]. The information of the sequential ordering of a program is used to amplify the amount of programs which are allowed while still guaranteeing determinism. SCL uses variables instead of signals, which are used by Esterel, for the primarily use of communication. By taking the sequential order of variable assignments in account, one variable can hold different values during a tick.

The SC MoC uses the initialize-update-read (iur) discipline, similar to ebt, which is used by Esterel. As the name suggests, first, initializations are done, followed by updates, and completing with reads. In the ebt discipline the first emission of a signal has to be done before any present test of the signal. Additional emissions in the current tick can be scheduled arbitrarily since they do not change the presence state of the signal. In the iur protocol, writes are handled as initializations by default, or updates. The order of updates can be scheduled arbitrarily. *Two variable accesses are `confluent` if their order of scheduling does not matter* [RSM+15]. While ebt applies to concurrent *and* non-concurrent accesses, independent of their sequential order, iur just applies to concurrent variable accesses.

Figure 1.4 shows an example for the scheduling with iur. The `fork par join` construct visualizes the creation of concurrent threads. As indicated by the comments, this example has four concurrent threads. All of them consist of one assignment and all want to access the variable `a`. Therefore, the order of the accesses has to be set by a scheduler. First, the initialization in thread 3 will be scheduled, followed by the updates of thread 1 and 4 in a random order, and lastly the read of variable `a` in thread 2 will be scheduled, resulting in `a` and `b` having the value 8. The variables will hold this value every time since iur guarantees determinism. The SCG (in-depth explanation in Section 1.2.1) in Figure 1.4 visualizes the before-mentioned example program and its dependencies. The statement at which an arrow points has to be scheduled after the statement at which an arrow starts.

Without iur, different values would be possible (race condition). For example, when scheduled in the order of the threads, `b` would hold the value 5, while `a` would hold the value 3.

```
fork
  a+=5  // thread 1
par
  b=a   // thread 2
par
  a=0   // thread 3
par
  a+=3  // thread 4
join
```

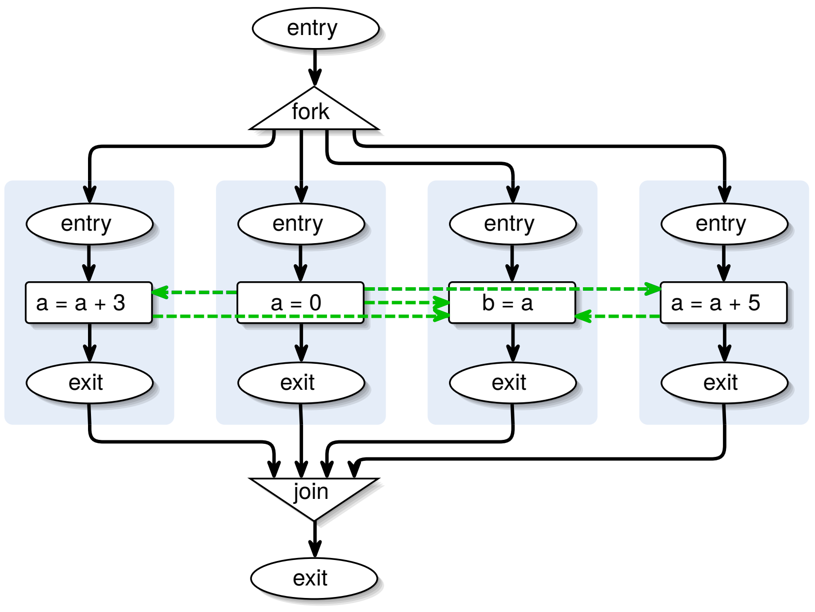


Figure 1.4. An example for the iur discipline in SCL and as SCG.

1.2.1 Sequentially Constructive Language

R. v. Hanxleden et al. proposed the Sequentially Constructive Language (SCL), designed as a minimal language [HMA+13] which is based on the SC MoC. For the description of the statements, the following letters are used. While **x** stands for a variable, **e** for an expression, **p** for a single or compound statement, **d** stands for a variable declaration. The eight statements of which SCL consists are the following.

fork p_1 **par** p_2 **join**	Parallel statement. Concurrent execution of p_1 and p_2.
l: p	A label l before a statement p.
pause	The execution is paused for the current tick.
$x = e$	Variable x gets assigned the value of expression e.
p_1; p_2	Sequence operator. Execute p_1 first and then p_2.
if e **then** p_1 **else** p_2	If expression e is true, execute p_1; otherwise, execute p_2.
goto l	Jump to Label l and continue the execution there. Label l has to be in the same thread as the goto statement.
{d; p}	A declaration d declares a variable for this scope and therefore the execution of p. The declaration overshadows outer declarations with the same name.

Sequentially Constructive Graph

The Sequentially Constructive Graph (SCG) is the graphical representation of SCL. While the solid black edges of the graph represent the control flow, the nodes represent SCL statements. Table 1.1 shows the mapping of SCL statements to SCG components. The beginning of a thread is displayed by an `entry` node and the end by an `exit` node, both of which are illustrated as an ellipse. While a triangle stands

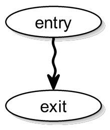
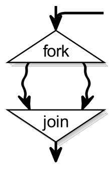
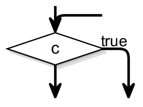
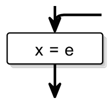
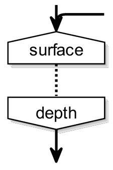
	Thread	Parallel	Sequence	Conditional	Assignment	Delay
SCL	`t`	`fork` t_1 `par` t_2 `join`	p_1 `;` p_2	`if (` c `)` s_1 `else` s_2	$x = e$	`pause`
SCG						

Table 1.1. Mapping from SCL to SCG [RSM+15].

for a `fork`, an inverted triangle displays a `join`. A rectangle stands for an assignment and a diamond represents an `if` statement. The tick boundaries are represented by a `surface` and `depth` node which are connected by a dotted line. Figure 1.4 shows an SCG with dependencies as mentioned previously.

1.2.2 Sequentially Constructive Esterel

Sequentially Constructive Esterel (SCEst) is an extension of Esterel. It was proposed by Rathlev et al. [RSM+15] and it includes the language features of Esterel v5 with SC semantics. It builds on SCL as its features are formally defined by transformations to SCL. SCEst extends Esterel in two ways. Foremost it uses the SC MoC, which reduces the number of rejected programs. A big difference to Esterel is its data handling since it allows variables to be shared and sequentially modified across threads. Therefore, the possibility to allow signals and their values to be reinitialized is given. This leads to an enhanced expressiveness of SCEst compared to Esterel and SCL. The second part is an extension of Esterel's statements by three new statements. As Esterel v5 is based on kernel statements to which all derived statements can be transformed to, SCEst makes use of this concept by reducing its Esterel statements and its new statements to its own set of kernel statements, which is SCL. The transformation rules are formally defined by Rathlev et al. [RSM+15]. In that paper all SCEst statements are wrapped up in the following five groups to provide a comprehensible structure.

control flow statements	`nothing`, `goto`, `pause`, `;` (the sequence operator), `loop` and `parallel`
signal handling statements	`pure local signal`, `pure emit`, `unemit` and `present`
data handling statements	`valued local signal`, `set` and `valued emit`
preemption statements	`suspend` and `trap/exit`
derived statements	`await` and `abort` in all their variants and `loop each`

For optimization, some transformations of derived statements are written as a direct transformation to SCL instead of transforming them to Esterel kernel statements first and the kernel statements to SCL statements second. The before mentioned three new statements are shown in the following:

unemit	The counter part to the `emit` statement, which sets the state of a signal to absent.
set	A valued signal is initialized with a given value.
goto	A restricted form of a standard goto. It jumps to the specified label or to the enclosing threads end label, when the first is not inside the thread.

Figure 1.5 displays the relation of Esterel, SCL and SCEst, as SCEst is an extension of Esterel but also based on SCL.

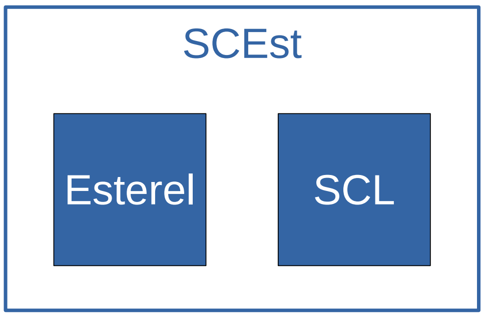


Figure 1.5. Illustration of the relation between SCEst, Esterel and SCL.

1.3 From Esterel to SCL

K. Rathlev presents an approach for a transformation from Esterel to SCL in KIELER in his master thesis [Rat15]. Rathlev showed that Esterel programs can be transformed to SCL programs while retaining their semantics and therefore that a smaller set of statements than the kernel statements of Esterel suffices to express Esterel programs. The transformation is done in one step by a structural approach. The transformation runs through all statements and begins its transformation at the bottom of the syntax tree (inside-out). Depending on the next enclosing statement, the corresponding transformation rule is used until a module is reached. For this approach it is essential to apply the transformation rules inside-out, since the outermost statement has priority over all inner statements, which leads to priority depended placements of conditionals after a pause statement for example, described in Section 5.5.2 in depth. This way the correct behavior of the resulting program is ensured. Since the whole program is given as an instance of the Esterel meta-model and the transformation produces an instance of the SCL meta-model, the transformation is an *exogenous* Model-to-Model (M2M) transformation, explained in Chapter 4 in detail. Mostly, the used transformation rules make it possible to apply the SC MoC to Esterel as the sequentially constructive structure is obtained by the transformation rules. Thus, many programs, which would be rejected by most Esterel compilers, are accepted.

Figure 1.7 shows the `ABRO` example from Section 1.1 in Esterel, before the transformation, and in SCL, after the transformation is done. Note that this `ABRO` example differs slightly from the previous one (`ABRO_derivedS`) to demonstrate the transformation of the `loop each` statement as an example and to clarify its meaning. A `loop each` statement is transformed to an `abort` and a `halt` statement inside a normal `loop` statement as shown in Figure 1.6. It is visible that the SCL program is more extensive than the Esterel program and also much less comprehensible. That lack of intelligibility motivated this thesis.

1.4 Single-Pass Language-Driven Incremental Compilation

The purpose of Single-Pass Language-Driven Incremental Compilation (SLIC) [Mot17] is to give a different approach to transform a source model to target code like C or Java. It performs an incremental compilation on a given source model, which consists of multiple M2M transformations in sequence. Its main characteristics are four in total. First, a specific transformation can only be used once on the model (*single-pass*). Second, a language feature can have multiple transformations at choice, but a transformation must only expand one feature. The transformation is so to speak an increment of the

```
module loop_each:
input R;
loop
  nothing;
each R;
end module
```

```
module loop_each_t1:
input R;
loop
  abort
    nothing;
    halt;
  when R;
end loop;
end module
```

Figure 1.6. `ABRO` transformation by Rathlev [Rat15].

compilation (*increment*). The second to last characteristic describes the order of the transformations, which is specific to the features of the language which should be transformed (*language-driven*). Lastly, after each transformation the result is a valid, self-contained model, ready to be inspected by the user (*intermediate models*).

A positive effect of SLIC is the possibility to inspect the intermediate model after each transformation and therefore being able to ease the process of certification for safety-critical systems. Also a complex transformation can be broken down into multiple transformations; therefore, the compiler is able to have a *divide-and-conquer validation and maintenance strategy* [Mot17]. Since the compilation is modularized, a language/compiler subsetting is made easier.

Figure 1.8 shows the general strategy of SLIC. Starting with a source model, an intermediate model will be generated, which is a valid, self-contained model, ready to be inspected by the user. Therefore, no extra information is saved outside of the model. After an arbitrary number of transformations and intermediate models respectively, the compilation ends in a fully transformed model, which can be C or Java code.

A *SLIC schedule* is a sequence of SLIC transformations, as shown in Figure 1.8. It is not trivial to determine that since the design of every single transformation has to be taken into account, which is explained in depth in Motika's dissertation [Mot17]. A transformation has the two properties *not-handled-by* and *produce* and can only expand one feature as mentioned before. A feature can be expanded by multiple transformations but at least one, which gives a user the choice, which transformation should be used. However, just one of the transformation alternatives can be executed.

Figure 1.9 shows properties for a SLIC schedule. It visualizes the feature f_1 which is expanded by the transformations τ_{f1} and $\tau_{f1'}$ for example (arrows with dotted lines). The outgoing arrows from $\tau_{f1'}$, labeled with `p`, indicate that $\tau_{f1'}$ may produce the features f_2 and f_3. Therefore, the transformation must be scheduled before the transformation of f_2 and f_3, since every feature can only be transformed once. The arrow with the dashed line, labeled with `nhb`, indicates that transformation τ_{f3} can not handle the feature f_1. Thus, feature f_1 must be transformed before f_3 since τ_{f3} is the only transformation alternative for f_3. The gray area marked as T_S displays the used transformations during this example for a SLIC schedule.

Considering Esterel as an example, each statement type can be a feature, but a statement can be transformed in multiple ways. For example, an `await` statement can be transformed to Esterel's kernel statements first and then later with an other transformation to SCL statements. It can also be transformed directly to SCL statements [RSM+15]. If both transformations were implemented, each would expand the `await` feature and therefore the user would have the choice which transformation

```
module ABRO:
input A,B,R;
output O;
loop
  abort
    [
      await A;
      ||
      await B;
    ];
    emit O;
    halt;
  when R;
end loop;
end module
```

```
module ABRO_SCL:
input bool A,B,R;
output bool O;
bool f_term=false;
fork
  l8:
  O = false;
  if !f_term then
    pause;
    goto l8
  end
par
  l7:
    fork
      l1:
      pause;
      if R then
        goto l4
      end;
      if !A then
        goto l1
      end;
      l4:
    par
      l2:
      pause;
      if R then
        goto l5
      end;
      if !B then
        goto l2
      end;
      l5:
    join;
    if R then
      goto l6
    end;
    O = O | true;
    l3:
    pause;
    if R then
      goto l6
    end;
    goto l3;
    l6:
  goto l7;
  f_term = true
join
end module
```

Figure 1.7. ABRO transformation by Rathlev [Rat15].

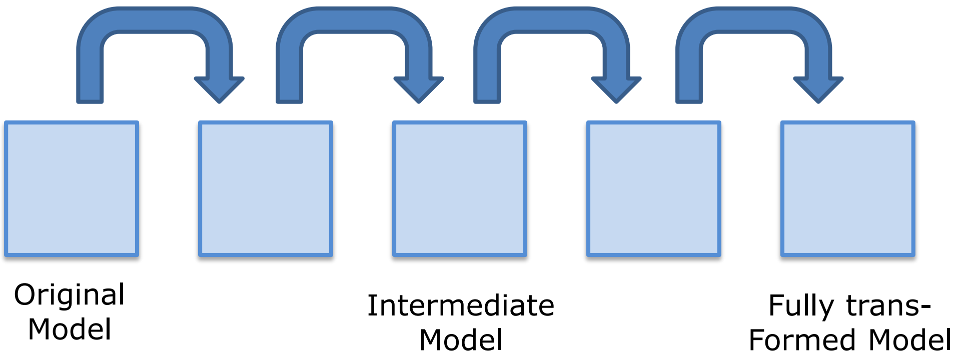


Figure 1.8. Single-Pass Language-Driven Incremental Compilation as an incremental model-based compilation strategy [Mot17].

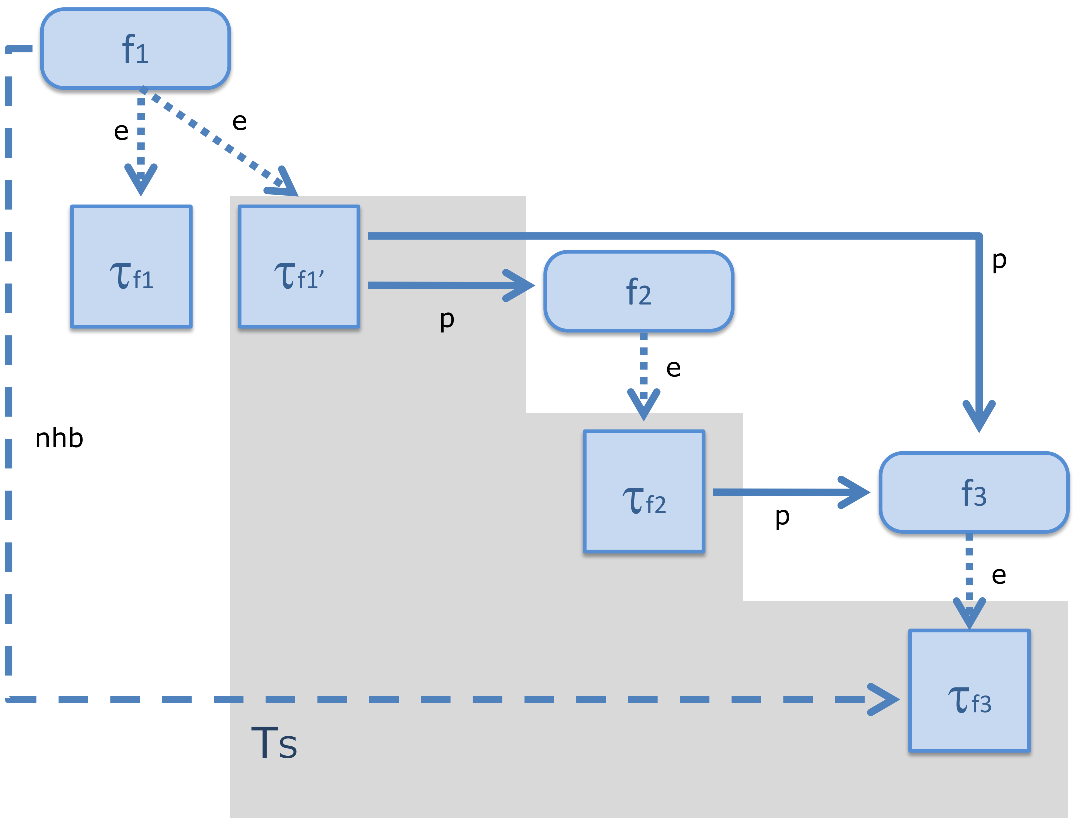


Figure 1.9. Example SLIC features f_1, f_2, f_3, SLIC transformations τ_{f1}, $\tau_{f1'}$, τ_{f2}, τ_{f3} and their **expand (e and dotted line), produce (p and solid line), and not-handled-by (nhb and dashed line)** order relations are shown. Also, a transformation choice T_S for a selection S of transformations is visualized [Mot17].

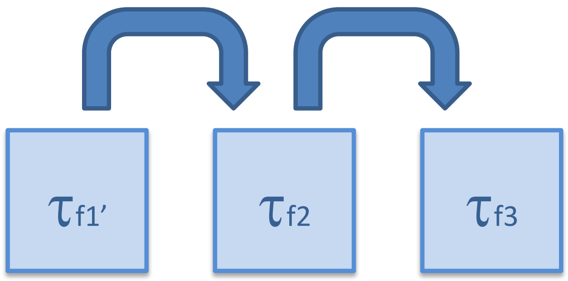


Figure 1.10. A SLIC schedule for the example of Figure 1.9 and the transformations $\tau_{f1'}$, τ_{f2} and τ_{f2} [Mot17].

should be used. However, in case of option one, the corresponding kernel statements would have to be transformed after the `await` statement.

To determine a sequence of transformations, a few questions have to be answered first. Does a feasible SLIC order exist? If it exists, how does the SLIC schedule look like? Lastly, does an *artificial production dependency* exist, which determines the order of the elimination of two features that do not have not-handled-by and produce dependencies?

1.5 Problem Statement

It has been shown by K. Rathlev [Rat15] that an Esterel program can be transformed to SCL, done in one single step using a structural approach, as mentioned before. The question remains if, following the SLIC approach, described in Section 1.4, it is also possible to transform a SCEst program step by step, producing intermediate self-contained models, which are valid by themselves with no information hidden. Those intermediate models could be inspected by the user. Therefore, the user would be able to understand the transformation of a single language feature better, which provides the opportunity to choose the right one for a given task. Also, the development or maintenance of the compilation chain would be easier since a single transformation could be easily replaced by a new one, or a possible error could be narrowed down to a single, more compact transformation and thus be fixed relatively fast.

This incremental compilation might seem as an easy modification of the structural approach, but it is by far not trivial, since the transformations can affect each other when they modify third statements. For example, when transforming the `Abort` and `Suspend` statements, conditionals have to be placed at tick boundaries. However, the transformation of those statements are independent and therefore it must be examined if the correct order of the conditionals can be retained.

1.6 Outline

In Chapter 2, used technologies for the implementation of the incremental compilation of SCEst are presented. The implementation of the step by step transformation is part of the KIELER project, which consists of Eclipse plugins. Furthermore, a general description of Eclipse including the usage of plugins and their structure, the grammar language Xtext, the programming language Xtend and the EMF are given.

In Chapter 3, two visual languages, the SyncCharts and the Sequentially Constructive Statecharts (SCCharts) for the synchronous MoC and the SC MoC respectively, are being explained as related work. Additionally, an overview of available Esterel compiler is given.

Chapter 4 presents the main part of this thesis. It is a description of the adjusted meta-model structures of Esterel and SCEst as well as the concept of the step by step transformation from Esterel/SCEst to SCL.

Building up on the previous structure, in Chapter 5, the implementation of the incremental compilation is described, as well as some features like code validation, formatting and scoping are presented.

It follows an evaluation of the implementation in Chapter 6 by displaying the step by step compilation of `ABRO` and discussing the size of the produced SCL code of some example programs. Additionally, emerged problems will be discussed.

It ends with a summary of the outcome of this thesis and possible future work in Chapter 7.

Chapter 2

Used Technologies

Since the implementation of the incremental compilation of SCEst is not a standalone one, other technologies were used, namely, Eclipse and KIELER, which are explained in this chapter to impart a basic knowledge.

2.1 Eclipse

The *Eclipse Foundation* released Eclipse[1] in 2001. Its intended use was an Integrated Development Environment (IDE) for Java, but since then, it has developed into an application for different areas [RB06]. It is an easy extendable system of plugins and offers the possibility to create a domain specific IDE.

The plugin system can be extended quite easily [Bol03]. Due to the plugins, a modular structure of projects can be achieved. An interaction can be defined as an *extension point* in a specific plugin and later used by other plugins. For example, Eclipse provides an extension point for new applications, so its appearance can be adjusted. One can decide which plugins should be used for a specific application when starting a new instance of Eclipse instead of loading all plugins.

In this section the Eclipse Modeling Framework, Xtext and Xtend will be explained to impart a basic knowledge of technologies used during this thesis, which is needed to understand Chapter 5.

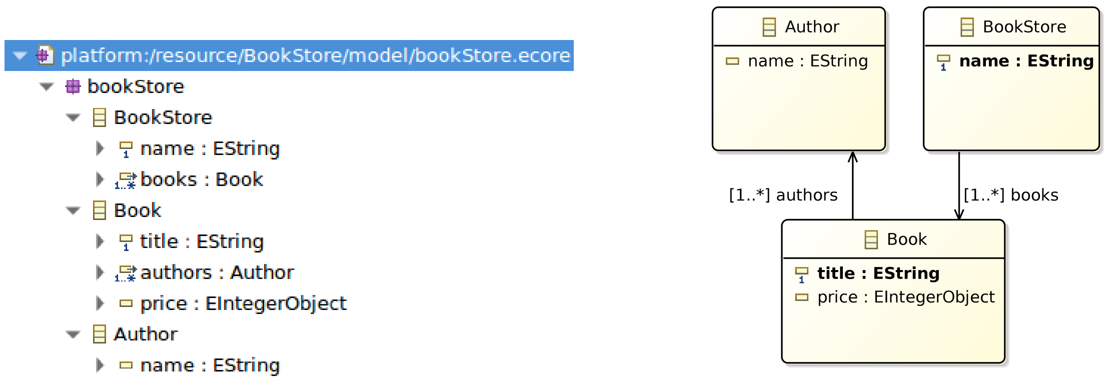
2.1.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF)[2] provides the possibility to declare models which structure is described in abstract meta-models. An *Ecore* model sets the layout of these meta-models. There are two editors, a *diagram editor* and a *tree editor*, for the creation and modification of a meta-model included in EMF. Figure 2.1 shows a book store as an example and the two different editors. Figure 2.1b shows the diagram editor and Figure 2.1a shows the tree editor.

In the tree editor, the `1` under the attribute `name` of `BookStore` means that a book store must have a name. The `1..*` under `books` stands for an arbitrary number but at least one. In contrast to the attribute `name`, `books` is a list of references to `Book` objects. Behind a colon the respective type is given. For `name` it is `EString`.

In the diagram editor, each square stands for an object type. Here they are `BookStore`, `Book` and `Author`. While the attributes of each object type are listed in the respective squares, a list included in an object is indicated by an arrow, which points to the type of object which it contains. The `[1..*] books` at the arrow from `BookStore` to `Book` means that a book store can have an arbitrary number of books but at least one, the same property compared to the tree editor since both editors represent the same model.

[1]http://www.eclipse.org
[2]http://eclipse.org/emf



(a) Ecore tree editor of the book store example. **(b)** Ecore diagram editor of the book store example.

Figure 2.1. The two different editors for the EMF meta-model.

2.1.2 Xtext

To be able to develop a Domain Specific Language (DSL) (e.g., programming languages in general), Xtext[3] is used, as it extends EMF. It is an open source framework. The Extended Backus-Naur Form (EBNF) is used in many cases to define the grammar of a language and likewise Xtext uses a similar form. Xtext can not handle left recursion since it produces LL-parsers by using the Another Tool For Language Recognition (ANTLR) parser generator and therefore most of the grammars can not be copied straight to Xtext. Left recursion means that a nonterminal symbol A exists in a given grammar which can be derived to $A\alpha$ where α is an arbitrary combination of nonterminal and terminal symbols [Pow99]. When reading from the left to the right, it can not be decided which rule should be applied. Since LL-parsers do exactly that, left recursion has to be removed in Xtext grammars. As mentioned, ANTLR is used to create a parser on the basis of a Xtext grammar. Other tools like an editor, code validation and highlighting are automatically generated, which are described in Chapter 5 to some extent, for an integration into Eclipse.

Going back to the previous example of a book store, Listing 2.1 shows its grammar definition in Xtext. To define a new book store, the keyword `store` has to be written. Followed by the name of the store, which is saved in the attribute `name`. The store must have one book but can have an arbitrary number of books, from 1 to infinity, which are separated by a comma. An arbitrary number, from 0 to infinity, is indicated by $*$, the second to last character in line 6, but since the term is preceded by `books+=Book`, which means that one book must be added, the arbitrary number is restricted to 1 to infinity. A book starts with the keyword `book`, followed by the title, multiple authors, but at least one author, indicated by `+` after `Author` in line 9, and a price. The price is optional as indicated by the `?` character. An author starts with the keyword `author` followed by the name.

2.1.3 Xtend

The programming language Xtend[4] is similar to Java. The discontinuation of unnecessary syntax overhead is the big difference to Java. Since Xtend compiles to Java code, it is possible to use a Xtend class in Java and a Java class in Xtend. Listing 2.2 shows an example method for creating a book store

[3]http://www.eclipse.org/Xtext
[4]http://www.eclipse.org/xtend

```
grammar de.cau.cs.kieler.BookStore with org.eclipse.xtext.common.Terminals

generate bookStore "http://www.cau.de/cs/kieler/BookStore"

BookStore:
  "store" name=ID books+=Book ("," books+=Book)*;

Book:
  "book" title=STRING authors+=Author+ price=INT?;

Author:
  "author" name=ID;
```

Listing 2.1. The book store grammar in Xtext.

```
1 def createBookStoreWithPredefinedBooks(EList<Book> oldBooks) {
2   var store = BookStoreFactory::eINSTANCE.createBookStore => [
3     name = "exampleStore"
4     books += oldBooks
5   ]
6 }
```

Listing 2.2. Creating a book store with a list of books in Xtend.

and adding a predefined list of books to it. The keyword `def` is used to define a method. The return type does not have to be declared explicitly since it can be derived. In this case the last object in this method is `store` and since it is of type `BookStore`, the return type of the method is also `BookStore`. Also the Java keyword `return` does not have to be written. In line 2, starting the declaration of the variable with the keyword `var`, the previously mentioned type of `store` is also not declared explicitly, but it can be derived from the method of the right hand side. The method `createBookStore` in line 2 was generated by the book store meta-model and returns an object of type `BookStore` as mentioned before. The newly created `BookStore` object is instantly initialized with the => operator in line 2 by passing the `BookStore` object into the lambda expression, indicated by the square brackets. In line 3, the name of the new store is set to `exampleStore` and the predefined list of books is added to the list of books of the new store, line 4.

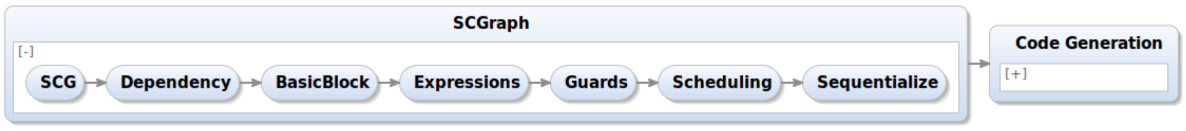


Figure 2.2. The KiCo View with the different transformations for an SCL program.

2.2 KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[5] is developed by the Real-Time and Embedded Systems group at Kiel University, Germany. It is a research project that focuses on designing complex systems with an improved graphical model-based approach. Two components, namely, KIELER Compiler and KIELER Model View, which are used during this thesis, are explained in this section.

2.2.1 KIELER Compiler

The KIELER Compiler (KiCo) is an Eclipse plugin and a framework providing transformations on EMF models using the SLIC approach as described in Section 1.4. It can be used by other plugins since it provides an extension point, described in Section 2.1, to register new transformations. A transformation itself has to be written in Xtend or Java. Therefore, it is possible to set up a hierarchical structure in which multiple transformations are ordered. This is done by defining dependencies.

Figure 2.2 shows a screenshot of the KIELER Compiler View, a graphical representation which is also integrated into KIELER, for an SCL file. It provides the possibility to transform an SCL program to target code like C or Java. Single transformations can be grouped for a better comprehensibility. In this specific KiCo View `SCGraph` would be an example of a group which includes the single transformations `SCG`, `Dependency`, `BasicBlock`, `Expressions` and so on. Each arrow displays a dependency. In this example the transformation `SCG` has to be done before every other transformation in the group `SCGraph`.

2.2.2 KIELER Model View

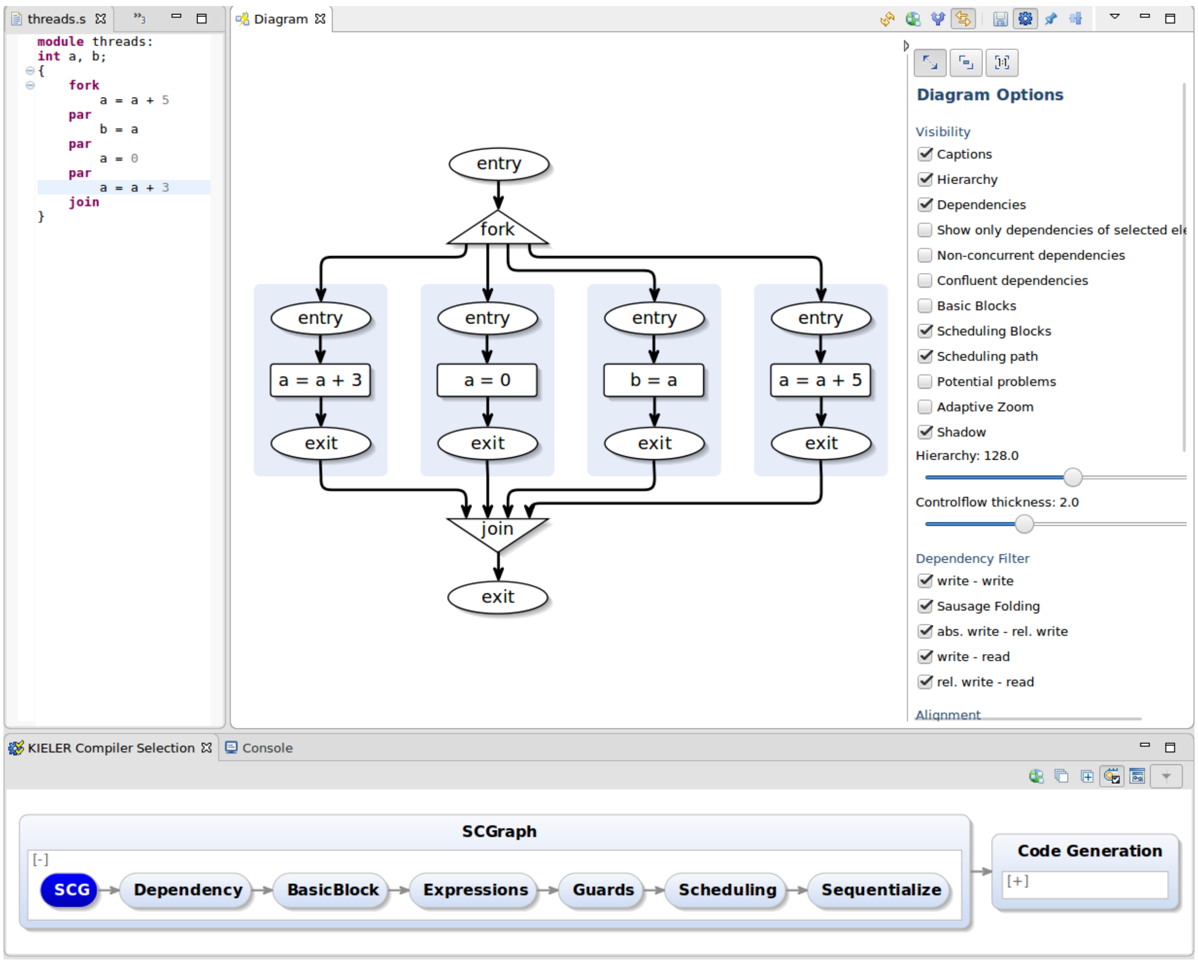
As KiCo, the KIELER Model View is an Eclipse plugin. It displays the current working model and can be a graphical representation, when possible, or just code. This is useful when applying a step by step transformation, since every step can be inspected in a clear way. Therefore, corrupt transformations can be identified a lot easier. The view is updated every time changes to the code are saved, another transformation is added, or a transformation is unselected.

As part of the KIELER Model View, KIELER Lightweight Diagrams (KLighD) is used to display any EMF model automatically. For this visualization KLighD is using a *diagram synthesis* which is specified in Xtend or Java [SSH13]. Key points are automatically providing an efficient way of browsing rendered diagrams, interactivity (e.g., highlighting) and the reduction of time used to create a graphical representation of a design.

Additionally, Eclipse Layout Kernel (ELK)[6] is used to provide a connection between diagram editors/viewers on the one hand and automatic layout algorithms on the other hand. A few standard layout algorithms are already included.

Figure 2.3 shows the example `threads` for the iur discipline, as mentioned in Section 1.2. Executing the transformation chain up to SCG, which is selected in the KiCo View, the KIELER Model View displays the SCG, as it is the representation of the current working model.

[5]http://rtsys.informatik.uni-kiel.de/kieler
[6]http://www.eclipse.org/elk



```
module threads:
int a, b;
{
    fork
        a = a + 5
    par
        b = a
    par
        a = 0
    par
        a = a + 3
    join
}
```

Diagram Options

Visibility
- [x] Captions
- [x] Hierarchy
- [x] Dependencies
- [ ] Show only dependencies of selected ele
- [ ] Non-concurrent dependencies
- [ ] Confluent dependencies
- [ ] Basic Blocks
- [x] Scheduling Blocks
- [x] Scheduling path
- [ ] Potential problems
- [ ] Adaptive Zoom
- [x] Shadow

Hierarchy: 128.0

Controlflow thickness: 2.0

Dependency Filter
- [x] write - write
- [x] Sausage Folding
- [x] abs. write - rel. write
- [x] write - read
- [x] rel. write - read

Alignment

Figure 2.3. The previous threads example for the iur discipline as an SCG with the KIELER Model View.

Chapter 3

Related Work

In this chapter related work that motivated this thesis is described. On the one hand, there are *SyncCharts*, which are a visual language for the synchronous MoC, and an interactive transformation from Esterel to the former is implemented in KIELER. On the other hand, there are SCCharts, which are a visual language for the SC MoC, and an interactive compilation of those using the SLIC approach is also implemented in KIELER. Additionally, a description of three different compiler for Esterel and their compilation approaches is given.

3.1 Visual Languages for the Synchronous MoC and SC MoC

In this section the two before mentioned visual languages for the synchronous MoC and the SC MoC, SyncCharts and SCCharts respectively, are described in the context of transforming Esterel to the former and the general compilation of the latter. Thereby, the motivation behind this thesis and the general approach of the incremental compilation of SCEst is being explained.

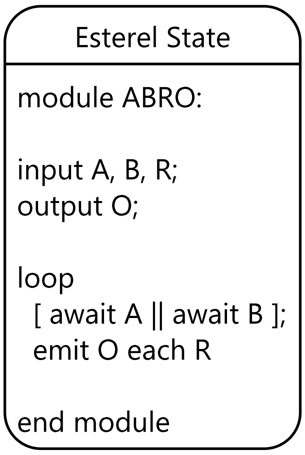
SyncCharts

SyncCharts are a graphical, synchronous language and representation of Esterel, since they are defined by a transformation from Esterel [PTH06]. They are based on the synchronous MoC. U. Rüegg covers in his bachelor thesis *Interactive Transformation for Visual Models* [Rüe11] a transformation from Esterel to *SyncCharts* [And03]. The transformation by Rüegg is implemented in KIELER and allows the user to visualize the transformation, which can be done step-wise. Each intermediate step can be inspected visually. In contrast to SCL, SyncCharts is not a minimal language and therefore provides the possibility of a direct transformation since almost all Esterel statements have a representing statement in SyncCharts. Thus, the step by step transformation covered in this thesis is more complex. However, the approach taken and its implementation are a motivation for this thesis. The transformation from Esterel to SyncCharts is done as follows.

Figure 3.1 shows the initial transformation on the `ABRO` example from Esterel to SyncCharts. This is done by creating a root macro state named `Esterel State` and copying the Esterel program's module in text form to the state's body. In addition, a reference to the actual module is saved in the state. Afterwards, the SyncCharts editor is opened and the user is able to select specific parts of the program to be transformed. When a selected transformation is done, a not yet transformed part of the program is copied in text form to the respective state's body, as shown with the initial transformation.

Therefore, this approach is quite similar to the one of this thesis. It produces a valid intermediate model after each transformation and lets the user choose which parts should be transformed. However, instead of allowing textual source code in the intermediate models, the incremental compilation of this thesis creates representing objects for every SCEst statement.

Figure 3.2 shows the `ABRO` example, described in Figure 1.2a, in SyncCharts. While states with a bold border are *initial*, the ones with double borders are *final*. The arrows with continuous lines are



Esterel State

```
module ABRO:

input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O each R

end module
```

Figure 3.1. Initial SyncCharts transformation of the `ABRO` example [Rüe11].

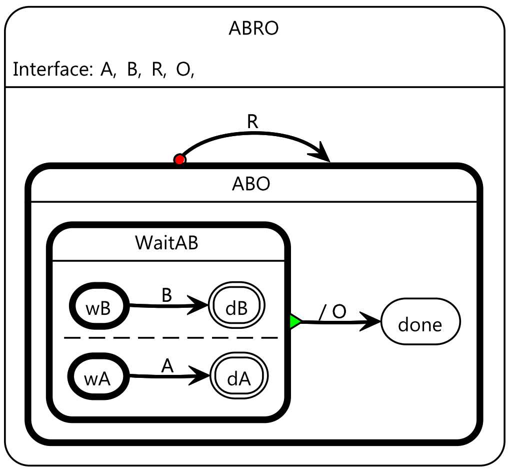


Figure 3.2. `ABRO` example representation in SyncCharts [Rüe11].

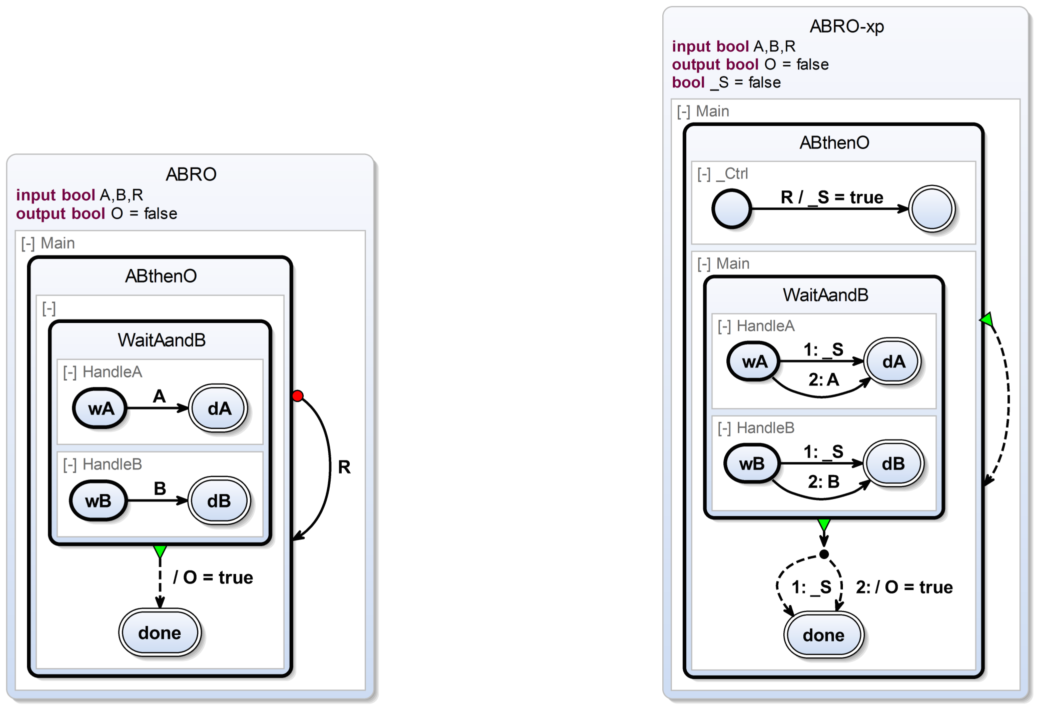
delayed transitions. Transitions have a *trigger* and an *effect*, separated by the `/` sign on top of a transition. In this example the trigger of the transition from `wB` to `dB` is `B`, while the effect of the transition from `WaitAB` to `done` is `O`. Note that the sign `/` can be omitted when the transition does not have an effect. A *normal termination* is indicated by a green triangle at the beginning of a transition, which is visualized in this example from `WaitAB` to `done`. Normal termination means that all parallel regions in the source state of a transition have reached a final state, here `dB` and `dA`. In this example `A` and `B` have to be present at some point after the initial tick. A *strong abort* is indicated by a red dot at the beginning of a transition. Here, the signal `R` triggers the strong abort of the state `ABO`, which leads to a reset of the state.

SCCharts

R. v. Hanxleden et al. proposed the visual language Sequentially Constructive Statecharts (SCCharts) [HDM+14], which is a conservative extension of SyncCharts and based on the SC MoC. The language features are separated into the *Core* and *Extended* SCCharts. While the core SCCharts *contain the key ingredients of statecharts, namely concurrency and hierarchy* [HDM+14], the extended SCCharts include all features of SyncCharts among others.

The compilation of SCCharts in KIELER uses the SLIC approach and is done by transforming the extended features to core features first, followed by a normalization, which makes the SCCharts ready to be transformed to an SCG. From there the implemented compile chain from SCG to target code is used. The transformation of signals leads to a transformation of all features that are dependent on a signal, and also to a rewrite of all expressions which reference the signals to the representing variables.

By using the SLIC approach, this compilation motivated this thesis's topic and the procedure is therefore quite similar to the step by step transformation of this thesis from Esterel/SCEst to target code over SCL. The latter differs from the former in not needing to transform all derived statements (resp. extended features) into kernel statements (resp. core features), since it uses direct transformation rules to SCL. This is done for optimization reasons, since this way the resulting code is more compact, e.g., shown for the `await` statement of Esterel [RSM+15], and additionally the compilation runtime can be reduced. Both, the compilation of SCCharts and the incremental compilation of SCEst of this thesis, are



(a) Extended SCCharts. **(b)** Core SCCharts.

Figure 3.3. ABRO example with and without extended features of SCCharts [HDM+14].

using the SLIC approach and therefore every single transformation specified is just used once during the compilation, the single-pass constraint of SLIC.

Figure 3.3a displays ABRO as an example for extended SCCharts. Despite a slightly different interface, it looks similar to the SyncChart of ABRO. A dotted transition is *immediate*. Figure 3.3b displays ABRO as an example for core SCCharts. It is the same program but only with core features. In the superstate ABthenO a second region _Ctrl is added, which sets the flag _S to true if R is present, to simulate the strong abort. If _S is true, a termination of ABthenO is triggered and immediately restarted with its immediate normal termination transition.

3.2 Esterel Compiler

Berry supervised a joint team at French Institute for Research in Computer Science and Automation (INRIA)[1] and École nationale supérieure des mines de Paris (ENSMP) starting the development of the official Esterel compiler. It has therefore the name INRIA Esterel compiler. While using the *circuit-* or *automaton-based* approach, the compiler compiles Esterel programs to either hard- or software. By using the circuit approach, the resulting netlist code as well as the needed time is almost linear to the source code size, which makes it advantageous for complex programs. However, the code is inefficient, since it also computes irrelevant results during the simulation of the circuit. The automaton approach generates a final state machine. The size of the code makes it only suitable for small programs, since it is exponentially bigger than the source code [Bt00]; however, it is much faster than the code of the

[1]https://www.inria.fr/

circuit approach. Both approaches admittedly do have intermediate formats which can be inspected by the user, but as stated in the manual:

The Esterel v5 91 compiler is made out of several internal processors controlled by a single esterel command. Normally, the user should only call this command and should never call directly the internal processors [Bt00].

By default, the intermediate code files are not given to the user. Whereas the approach of thesis uses SCL/SCG as an intermediate format ready to be inspected and makes use of the compilation chain from SCL/SCG to target code.

Edwards et al. started in 2001 the development of the open source Esterel compiler Columbia Esterel Compiler (CEC)[2] at the University of Columbia [PEB07]. Its intended use was for research in generating hard- or software from Esterel. The CEC can be used to generate a C Program from Esterel. In addition, it can generate a Verilog or Berkeley Logic Interchange Format (BLIF) circuit description. This can be done by three different methods, namely, the Program Dependence Graph (PDG), the dynamic list and the virtual machine code generation, which are explained in the article *Code Generation in the Columbia Esterel Compiler* by Edwards et al. [EZ07]. The CEC uses a variant of Potop-Butucaru's [Pot02] graph code (GRC) as the intermediate representation and supports just a subset of Esterel v5. The GRC is quite similar to an SCG and therefore the CEC compilation is like the compilation of SCEst to target code of this thesis, since both have a similar intermediate representation, GRC and SCG respectively. An SCG has additionally multiple kinds of dependencies and allows an arbitrary control flow with loops.

Another compiler project is the Freesterel[3] compiler proposed by Coadou and Ferrero at INRIA for education and research purposes by cloning the genuine Esterel Compiler, since no other free Esterel v7 compiler exists, but as of now, no finished compiler or updates have been presented. As it was proposed in 2010, one can assume that the project has stopped. This compiler was meant to give a user just the target code as the INRIA compiler and therefore is different to the incremental compilation of SCEst of this thesis.

A more detailed but also compact explanation of the compilers and especially of the GRC and the PDG is given in Rathlev's master thesis [Rat15].

[2]http://www.cs.columbia.edu/ sedwards/cec
[3]http://freesterel.sourceforge.net/

Chapter 4

Incremental Compilation

The difference between Rathlev's structural approach and the step by step approach of this thesis is that in the structural approach a given Esterel program will be transformed completely in one step to an SCL program. The transformation starts at the innermost statement and after that the next enclosing statement will be transformed and so on. The process stops when the next enclosing object is the root object of the program. So for every possible statement there is a transformation ready to be used. Since everything is done in one step, it is not clear which statement in the source program led to which statement or statements in the target program. In case of a slightly more comprehensive source program, the transformation itself is therefore not as easy to understand as it could be.

Figure 4.2a displays the user experience when the traditional compilation approach is used. The compiler gets a program in text or graphical model form, performs its transformations, and returns the program in target code. In this case the compiler is a black box that gives no inside on its intermediate results. When the user does not have the knowledge to understand the intermediate results, this is not a problem.

Figure 4.2b shows an interactive compilation using SLIC. It is a white box compiler, which displays intermediate results to the user. The interactive compiler takes additionally an interactive user compiler selection besides the source model as input. Therefore, the user is actively able to modify the compiler. Thus, as described in Section 1.4 , SLIC allows the user to apply a self-chosen set of transformations to the source model. A few advantages of SLIC as an interactive compilation compared to the traditional compilation are the following. Since the user is able to see intermediate results, which means that a feature is displayed by an other more basic feature, the understanding of the semantics of language features is improved. Further, the knowledge of the models is improved since it is more likely that a user is able to understand the language features being used while seeing intermediate models. Therefore, the user can realize how to use language features in a correct way, which eases the development of correct safety-critical systems, which depends on a reliable model and compiler. Additionally, the user might have a set of multiple transformations for one feature. Depending on the wanted result, the user is able to pick the best transformation to accomplish it. Moreover, the development of the compiler is easier, since it is modularized. If an error is found, it is most likely easier to fix it, since it is already known in which specific transformation it occurs. Also, a transformation of a feature can easily be deleted and replaced by a new one.

The other side of the coin is the probable deceleration of the compilation, since a run through the whole program for every transformation is initiated in contrast to one run-through. However, weighing the benefits against this drawback, the interactive SLIC compilation is better suited for designing safety-critical systems, since it provides a more reliable compiler and models.

Figure 4.1 displays the relation of Esterel, SCL and SCEst and the strategy for the incremental compilation of SCEst used in this thesis. In Section 4.1, the structure of the underlying SCEst meta-model, which represents a SCEst program during a compilation, will be explained in detail. But for now it suffices that in Figure 4.1 SCEst is displayed as a superset of Esterel and SCL. This means that the implementation of the model of SCEst not only includes the features stated in its formal definition but also all SCL statements. This implementation simplifies the incremental compilation of SCEst as

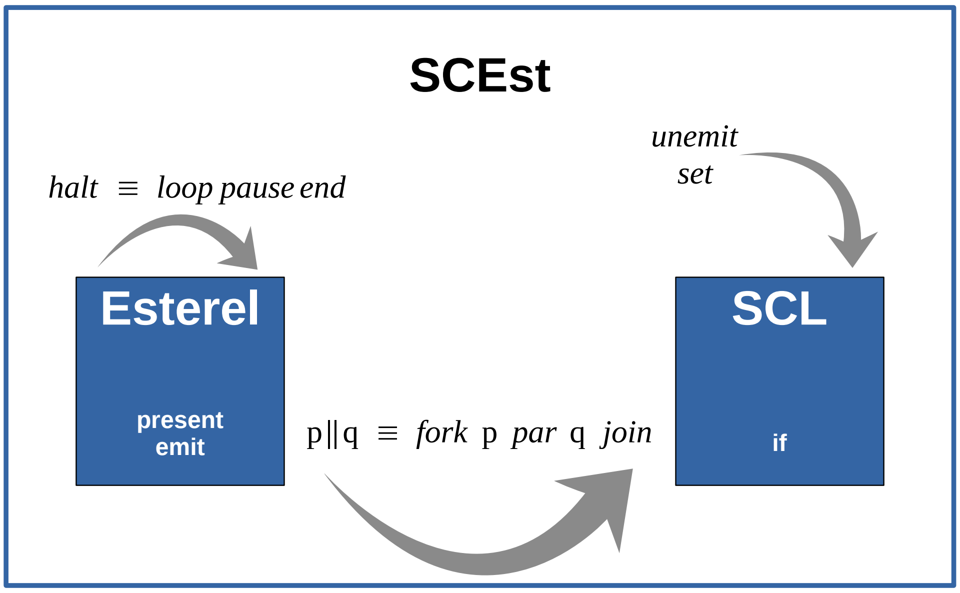


Figure 4.1. Illustration of the relation between SCEst, Esterel and SCL.

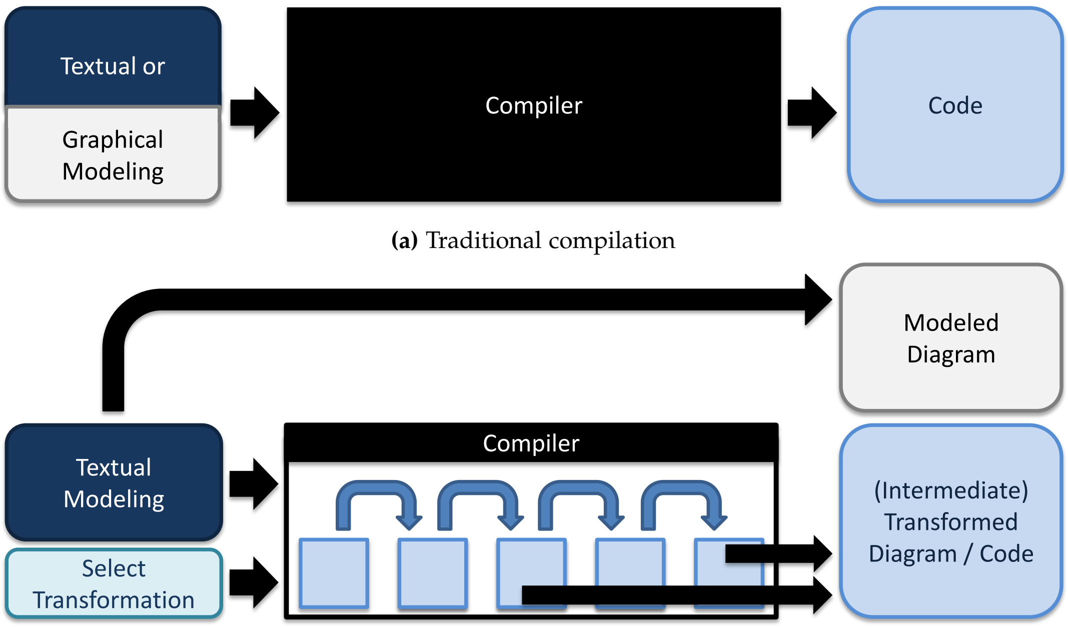


Figure 4.2. Difference for modeling using traditional compilation and interactive model-based compilation [Mot17].

described later in detail. For example, while the transformation of the `halt` statement will be within Esterel's statements, since it is a derived statement which is transformed to Esterel's kernel statements `loop` and `pause`, the transformation of the `parallel` statement leads to a `fork par join` construct of SCL. The new SCEst statement `unemit` will be transformed directly to a representation in SCL. However, during the compilation of an arbitrary statement, the scope of SCEst is not left and therefore the transformations can be done on one underlying model, which benefits will be explained in depth later.

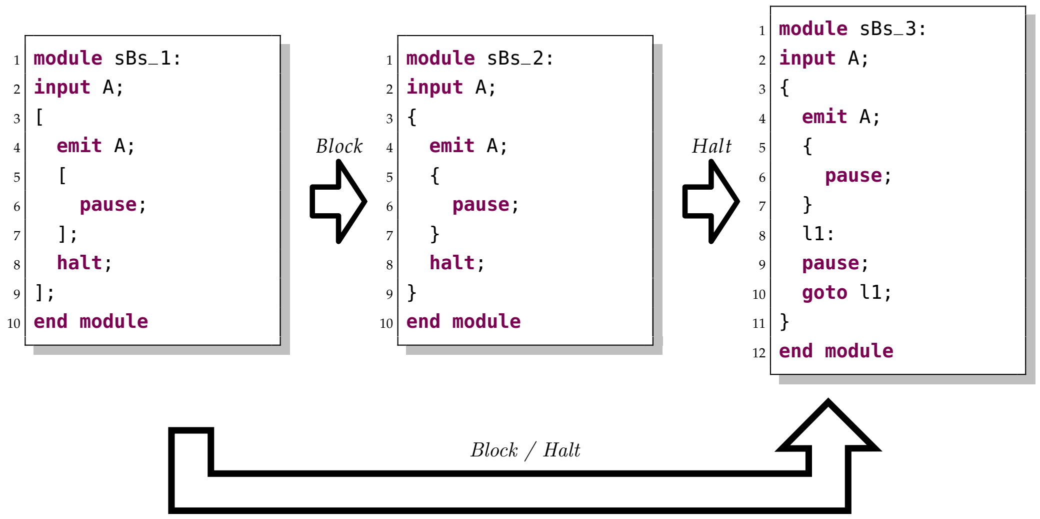
The transformation of a SCEst program to an SCL program is done by transforming a given instance of the SCEst meta-model to an instance of the SCL meta-model. Only if all Esterel/SCEst statements have been transformed to SCL statements, the transformation to the SCL meta-model is done. The single transformations of the different SCEst statements are done on an instance of the SCEst meta-model. Thus, there are two different kinds of Model-to-Model (M2M) transformations, the *exogenous* and the *endogenous* M2M transformation [MG06]. An exogenous transformation has different target and source meta-models. In contrast, an endogenous transformation has the same target and source meta-model.

To implement the incremental compilation of SCEst in KIELER, the following strategy is used. The incremental compilation of SCEst is divided into the following features needed for the SLIC approach, as described in Section 1.4. 27 features represent each an Esterel statement, which has to be transformed. One feature provides an initialization mechanism, which is described in Section 5.5.2 in detail, an other feature provides an optimization for the chosen names of labels, and lastly a feature is used to do the exogenous M2M transformation from an instance of the SCEst meta-model to an instance of the SCL meta-model. Each feature has just one transformation. Therefore, the possibility to choose between alternative transformations for a given feature, as described in Section 1.4, is not provided. After each transformation the intermediate model is a valid one and can be inspected by the user by using the KIELER Model View, described in Section 2.2.2. Some transformations have not-handled-by and/or produce dependencies, which give KiCo information on the order in which the transformations have to be scheduled. If KiCo can choose between different features, since their dependencies are not on each other, it can choose an arbitrary order, which can be different for every execution of the compilation chain.

As explained, the step by step approach's purpose is to provide a transformation which is understandable for the user. It allows the selection of a specific kind of a statement, which then will be transformed for the whole program. The example in Figure 4.3 shows the transformation of all `Block` statements. If the `Block` statement is selected to be transformed, a run through the whole source program is triggered in which every time a `Block` statement is found, the `Block` statement will be transformed to a `ScopeStatement`. After the transformation the user is able to look at the produced code and thus can realize how a specific statement is being transformed and therefore is able to understand the transformation (the second listing in Figure 4.3). When additionally another type of statement is selected as displayed with the `Halt` statement in Figure 4.3, the transformation for both statements is done in one big step from the viewpoint of the user, consisting of the two separate transformations. The order is dependent on the constraints of the selected transformations. Afterwards, the result is shown to the user (third listing in Figure 4.3). Note that in this mini example the transformation of the whole program at once is understood quite easily, but for more complex ones it is not.

To be able to do these transformations in KIELER, first the formally defined language SCEst [RSM+15] has to be implemented and integrated in KIELER and since it is an extension of Esterel and therefore those two implementations are closely connected, the already integrated Esterel grammar and meta-model are updated as well. The single transformation rules are already defined and described [Rat15; RSM+15]. However, they are not complete and therefore a few modifications of the rules are explained in this chapter.

For accomplishing the implementation of the incremental compilation of SCEst, the best possible



```
module sBs_1:
input A;
[
  emit A;
  [
    pause;
  ];
  halt;
];
end module
```

```
module sBs_2:
input A;
{
  emit A;
  {
    pause;
  }
  halt;
}
end module
```

```
module sBs_3:
input A;
{
  emit A;
  {
    pause;
  }
  l1:
  pause;
  goto l1;
}
end module
```

Figure 4.3. The different results which a user can inspect: While just the `Block` statement being selected leads to the second listing, the `Block` and `Halt` statement being selected leads to the third listing.

structure of the different meta-models has to be found, which is done in Section 4.1. Additionally, it must be seen how the incremental compilation of SCEst can be implemented to follow the principles of SLIC, described in Section 1.4, which is done in Section 4.2.

4.1 Structure of Meta-Models

There are a few possibilities to structure the implementation of the SCL, Esterel and SCEst meta-models. Figure 4.4 shows the previous structure of SCL and Esterel and their needed components. While SCL uses expressions of the *KExpressions* plugin, which is used by multiple other plugins, Esterel uses its own implementation of KExpressions, since it needs more components than the ones provided by KExpressions, for example, `sensors` are used in Esterel but not provided by KExpressions.

Therefore, three different approaches on how to structure the meta-models are explained in this section. The main objectives are providing a structure, which is as lean as possible by making use of already implemented features, e.g., the fusion of the two KExpressions implementations, providing the possibility to have intermediate models, as explained in Section 1.4, and making the transition to an instance of the SCL meta-model as easy as possible.

4.1.1 Separate Meta-Models for Esterel and SCEst

A first approach is shown in Figure 4.5a. The previous structure is adopted and the new meta-model of SCEst is added as an extension of the Esterel meta-model. This is a simple approach since nothing would have to change and just a new small meta-model needs to be added. However, the simple implementation of the meta-models leads to a more complex implementation of the transformations, in particular the last exogenous M2M transformation. Every instance of an expression in SCEst would have to be transformed to a corresponding instance of a KExpression. Since multiple SCL statements are not included in the formally defined language SCEst, it leads to a limitation of possible single transformations within the SCEst meta-model. For example, the `Block` and `EsterelParallel` statements

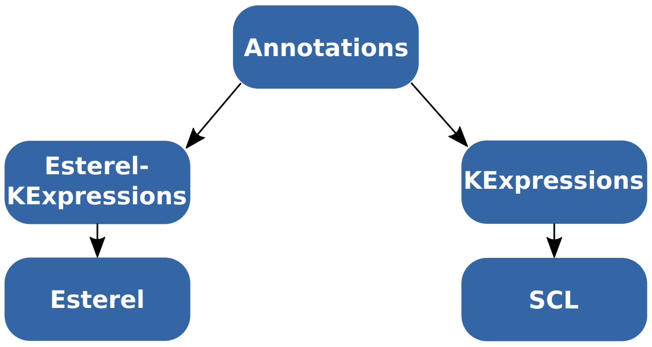


Figure 4.4. The previous structure of the meta-models, needed for the incremental compilation of SCEst.

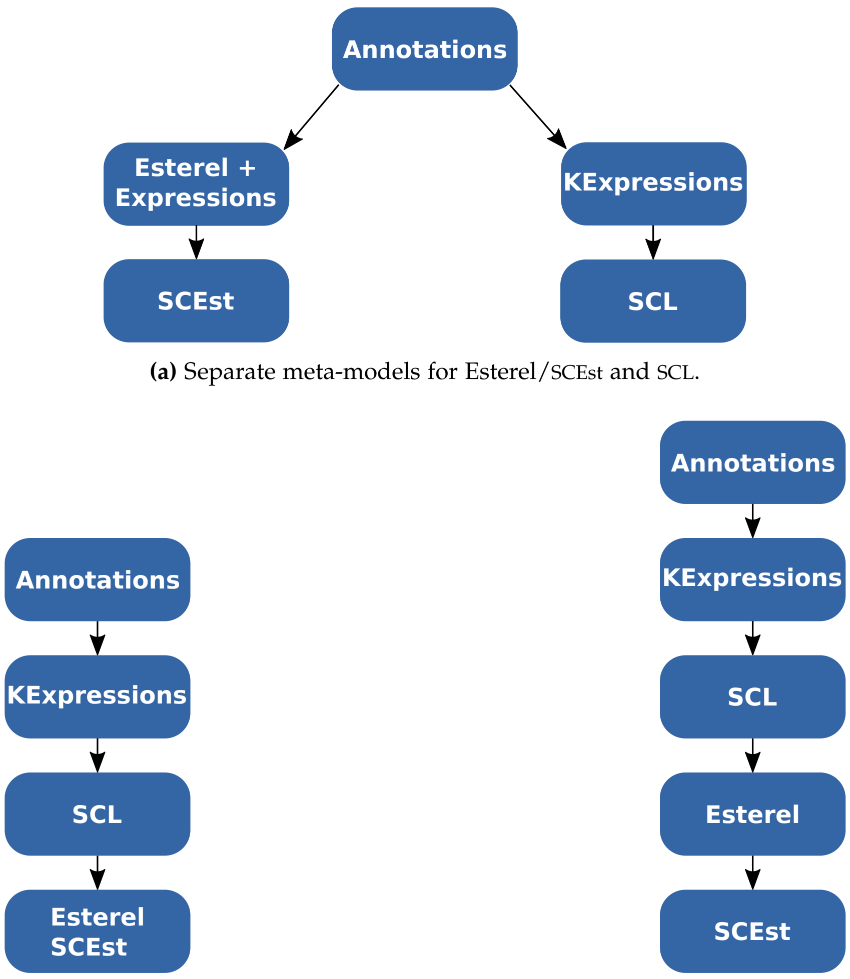
of Esterel as well as the transformation of the signals would have to be done together in the last step, the exogenous M2M transformation. Since it does not serve a good step by step transformation, it leads to the following approach of a combined meta-model for Esterel and SCEst.

4.1.2 New Structure with One Meta-Model for Esterel and SCEst

The second approach is shown in Figure 4.5b. It is just one meta-model for Esterel and SCEst which extends SCL. Thus, it is connected to the definitions in SCL. Since SCL extends KExpressions, SCEst can also use those definitions. Having a combined meta-model for Esterel and SCEst would lead to the following problem. If users want to write a pure Esterel program, there has to be a mechanism which makes sure that only Esterel statements are accepted. Therefore, a special comment at the beginning of the program could be used, so the editor can recognize if just pure Esterel statements are allowed or SCEst and SCL statements are allowed also. For example, the comment `%SCEst` could be placed in the first line of a SCEst program and if the first line does not include this comment, only pure Esterel statements are allowed by default. If a user enters a SCEst statement without having placed the comment, the editor would display an error and would not accept the program. Since this approach would primarily use KExpressions and therefore the structure of the existing Esterel and its own KExpressions meta-model and grammar would need to be adjusted, this approach would not be as simple as the first one in implementing the meta-model, but it would make the implementation of the step by step transformation easier because not every expression would have to be transformed as in the first approach. Since all SCL statements are known, it is possible to allow also SCL statements in this new meta-model of Esterel and SCEst, which is described more detailed in the following approach. Those advantages makes this approach superior to the first one for the task of implementing a step by step transformation.

4.1.3 New Structure with Two separate Meta-Models for Esterel and SCEst

The third approach is shown in Figure 4.5c. It differs from the second approach in the way of a distinction between the Esterel and the SCEst meta-model. Like the previous approach, the Esterel meta-model extends the SCL meta-model. But now a new meta-model, the SCEst meta-model, extends the Esterel meta-model. Therefore, the Esterel meta-model could use defined components of SCL, but it does not. Only in the SCEst meta-model the definitions of SCL are being used. The implementation of the SCEst meta-model and grammar differs from the formally defined language SCEst. Not only Esterel statements plus the three added statements of SCEst are included in the implementation of SCEst, but



(a) Separate meta-models for Esterel/SCEst and SCL.

(b) One meta-model for Esterel and SCEst.

(c) Two separate meta-models for Esterel and SCEst.

Figure 4.5. The structure of the three approaches of the meta-models.

also all SCL statements are allowed. This leads to the benefit of being able to transform every statement on its own and not like the first approach having to do multiple transformations in one step. For example, all `Block` statements of an Esterel program can be transformed to `ScopeStatement` statements of SCL and they can be represented in the SCEst meta-model. Also the Esterel and the SCEst meta-model make use of the KExpressions. Just the expressions which can not be represented with KExpressions have their own definition in the Esterel meta-model. The Esterel meta-model is more comprehensive since it is not a minimal language as SCL. The SCEst meta-model is very small because it extends the Esterel meta-model only by two statements, the `set` and `unemit` statement, and the SCL statements. The latter are obviously already defined in the SCL meta-model. Because of having two meta-models for Esterel and SCEst, the disadvantage of the previous approach, the programmatically distinction between an Esterel and a SCEst program, is dissolved. In addition, by having a distinction between those two meta-models, the structure is clearer. Out of the three approaches this third approach is therefore the most suited for the task of implementing a step by step transformation.

4.2 Transformations

The structural approach for a transformation of an Esterel program to an SCL program has been implemented in KIELER by Rathlev in his master thesis [Rat15]. While most transformation rules are already described there, Rathlev et al. proposed SCEst [RSM+15] with additional transformation rules. This section describes a fraction of those transformations as well as some modifications to the transformations needed for the incremental compilation with the SLIC approach.

For a better understanding the following short forms are used.

p	A single or compound statement.
l<number>	A generated unique label. For example **l1**, **l2** or **l99**.
gotoj l3	A restricted form of goto, which jumps either to **l3** if the label is not outside its enclosing thread or to the surrounding threads end label.
p $[s_1 \rightarrow s_2 \mid s_3 \rightarrow s_4]$	Every s_1 in **p** is replaced by s_2 and every s_3 by s_4.

4.2.1 Existing Transformation Rules

For a better understanding, an extract of the transformation rules, already defined by Rathlev [RSM+15], are repeated in this subsection. The ones mentioned are on one hand the transformation rules for the new SCEst statements `unemit` and `set` and on the other hand the transformation rules used for the `ABRO` example for which a step by step transformation is shown later.

UnEmit

```
unemit A
```

⇨

```
A = false;
```

Listing 4.1. Transformation of an unemit.

An `UnEmit` statement is transformed straight forward to an absolute write to the corresponding variable, as shown in Listing 4.1. The absolute write ensures that `unemits` are scheduled before `emits`.

Pure Emit

```
emit A
```

⇨

```
A = A || true;
```

Listing 4.2. Transformation of a pure emit.

An `Emit` statement is transformed straight forward to a relative write to the corresponding boolean variable, as shown in Listing 4.2. The relative write is used to ensure that every `UnEmit` statement is executed before `Emit` statements.

Valued Emit

```
1 emit A(v)
```

```
1 A = A || true;
2 A_cur = f(A_cur, v);
```

Listing 4.3. Transformation of a valued emit.

A valued emit is transformed to a relative write of the corresponding boolean variable, line 1 in Listing 4.3, and the new value is combined with the current signal value and the combine function f and saved in `A_cur`, line 2.

Set

```
1 set A(v)
```

```
1 A = A || true;
2 A_set = A_set || true;
3 A_cur = v;
```

Listing 4.4. Transformation of a `Set` statement of SCEst.

As shown in Listing 4.4 line 3, the transformation of a `Set` statement differs from the `Emit` statement in that it replaces the relative write to `A_cur` with an absolute write to `A_cur`. Therefore, all `emits` are executed after any concurrent `set`.

Halt

The `halt` statement stops the execution permanently. As shown in Figure 1.1, the `halt` statement is transformed to a label, followed by a `pause` statement and lastly a `goto` statement, which initiates a jump back to the label.

Parallel

```
p
||
q
```

```
fork
  p
par
  q
join
```

Listing 4.5. Transformation of a `Parallel` statement of Esterel.

The `Parallel` statement of Esterel stands for concurrency. The transformation to SCL is straight on since the `||` operator is replaced by a `fork par join` construct, as shown in Listing 4.5, where `p` and `q` are concurrently executed.

Loop

```
loop
  p
end
```

```
l1:
p;
goto l1;
```

Listing 4.6. Transformation of a `loop` statement of Esterel.

The `loop` statement is transformed straight forward as well to a label before the loop's body and a goto after the body, which initiates a jump back to the label, shown in Listing 4.6.

Delayed Await

```
await A
```

```
l1:
pause;
if !A {
  goto l1;
}
```

Listing 4.7. Transformation of a delayed `await` statement of Esterel.

The delayed `await` statement allows for the execution to continue only if a specified signal is present after the initial tick. Therefore, as shown in Listing 4.7, it is transformed to a label, followed by a `pause` statement and a conditional after the pause, which checks for the previously mentioned signal not to be present. If the signal is not present, a `goto` statement is executed which initiates a jump back to the label; otherwise, the execution continues.

Strong Delayed Abort

```
abort
  p
when A
```

```
{
  bool abort_flag = false;
  p [ pause  -> pause; if A {abort_flag=true; gotoj l1;}
    | join   -> join; if abort_flag {gotoj l1;}]
  l1:
}
```

Listing 4.8. Transformation of a strong delayed `abort`.

A strong delayed `abort` indicates a preemption of the execution of its body when its condition holds true. It is transformed as shown in Listing 4.8. The body, in this case `p`, is transformed in the way that after each `pause` statement a conditional is added, which therefore checks at the beginning of each tick, except the initial one, if its condition holds true, and if so, an abort flag is set to true, here `abort_flag`, and a `goto` statement is executed, which initiates a jump to the label placed after `p`. If the `goto` statement is inside a thread, which is inside `p`, the `goto` initiates a jump to the thread's end label. Therefore, the additional flag is needed, which is tested after each `join` to determine if a thread terminated by a preemption, and if so, a jump to the label after the body or to the enclosing thread's end label is initiated.

4.2.2 Preserving Declarations for Incremental Compilation

The following transformation rules are a slight modification of the already defined ones. The only difference is, for the incremental compilation the old signals are being kept while creating new variables for their representation in SCL since other statements depend on them. For example, the `Emit`, `UnEmit` and `Set` statements need an actual signal. In case of the deletion of the old signals, all of them would need to be transformed as well, but this would lead to a coarser partition of single transformations. Additionally, this would come at the cost of not being able to create a set of features, each representing a statement of SCEst, and transforming each feature just once during the compilation, which is also part of the SLIC approach. To provide this finer distinction, the signals are being kept until the last exogenous M2M transformation to an instance of the SCL meta-model.

Signals

```
module inputExample:
input A;
p
```

⇨

```
module inputExample_t:
input A_sig;
{
  input bool A;
  p;
}
```

Listing 4.9. Transformation of an input signal.

The initial state of an input or inputoutput signal in each tick is set by the environment. The input signals are transformed straight forward to boolean variables.

```
module outputExample:
output B;
p
```

⇨

```
module outputExample_t:
output B_sig;
{
  output bool B;
  bool f1_term;
  fork
    l1:
    B = false;
    if (! f1_term) {
      pause;
      goto l1;
    }
  par
    p;
    f1_term = true;
  join
}
```

Listing 4.10. Transformation of an output signal.

An output signal is absent by default in every tick, which means the initial value of the corresponding boolean variable has to be set to false in every tick. Creating two threads, with the first being used to do that and the second being used to execute `p`, is a solution to this problem. After the execution of `p`, the boolean variable `f1_term` is set to true to indicate that `p` is completely executed and therefore the first thread, which is setting the initial value of the representation of the output signal, can be terminated. Note that `f1_term` is a flag with an arbitrary number. If an other termination flag is needed, e.g., for a local signal, it would be named `f2_term`. In the first thread the boolean variable, which represents the output signal, is set to false by an absolute write, following by a check of the variable `f1_term`. If `p` has been executed completely and therefore `f1_term` is true, the thread terminates, and if not, the thread pauses and starts the same execution again in the next tick. Note that the absolute write is scheduled before a relative write, as described by Rathlev [RSM+15], and therefore setting the output signal's corresponding boolean variable to false is scheduled before any emit, which is done with a relative write.

A local signal, as shown in Listing 4.11, is transformed in the same way as an output signal, except that in line three of the second listing the representing boolean variable is not declared as `output` and the local signal still surrounds the transformed body.

```
1 signal B in
2   p
3 end signal
```

⇨

```
 1 signal B_sig in
 2   {
 3     bool B;
 4     bool f1_term;
 5     fork
 6       l1:
 7       B = false;
 8       if (! f1_term) {
 9         pause;
10         goto l1;
11       }
12     par
13       p;
14       f1_term = true;
15     join
16   }
17 end signal
```

Listing 4.11. Transformation of a local signal.

Valued Signals

```
1 signal A := A_init :
2   combine TYPE with f in
3     p
4 end signal
```

```
 1 signal A_sig := A_init :
 2   combine TYPE with f in
 3     {
 4       bool A, A_set;
 5       bool f1_term;
 6       TYPE A_cur, A_val = A_init;
 7       fork
 8         l1:
 9         A = false;
10         A_set = false;
11         if (! A_set) {
12           A_cur = f_neutral;
13         }
14         if (A) {
15           A_val = A_cur;
16         }
17         if (! f1_term) {
18           pause;
19           goto l1;
20         }
21       par
22         p;
23         f1_term = true;
24       join
25     }
26 end signal
```

Listing 4.12. Transformation of a valued local signal.

A valued signal will be transformed into two boolean variables and two variables of the signal's type, shown in Listing 4.12. Note that `TYPE` is a placeholder for a type in Esterel, for example `integer` or `boolean`. The first boolean variable `A` displays if the signal itself is absent or present, with being false or true, like the transformation of a pure signal. The second variable `A_set` is set to true when a `Set` statement of SCEst is executed in this tick. Its initial value is false in every tick, like the first boolean variable. The first variable of the signals type, `A_cur`, holds a temporarily value, which will be explained in a moment. The second variable of the signals type, `A_val`, holds the actual value of the signal. Since the value is not reseted in each tick, the value in `A_val` must be the same for the next tick if no valued emit, unemit or set occurs in the current tick. `A_cur` is set to the neutral element `f_neutral` of the combine function `f` in each tick, second listing line 12, if no `Set` statement is executed on `A` in the current tick. So in case of an emit of `A`, the new value is combined with the current signal value. But as mentioned before, the value of the signal should stay the same across ticks if no operation occurs on the signal. To solve the problem of overwriting the value of the signal with the neutral element, the new value is build in `A_cur` and copied to `A_val` only if an emit occurs. A reference to the value of a signal (`?A`) is replaced by `A_val`.

Chapter 5

Implementation

In this chapter the implemented features that are needed for the incremental compilation of SCEst are described. The *Annotations*, *KExpressions*, *KEffects*, *KExt* and SCL implementation were already part of KIELER, but for a better understanding they are described in the following. Within this thesis the implementation of Esterel in KIELER was modified, as explained in Chapter 4. Additionally, SCEst and the incremental transformation were implemented in KIELER during this thesis. The hierarchical structure of the meta-models can be seen in Figure 5.1. Note that the structure differs from the one in Figure 4.5c, because for the previous chapter, it was not needed to display the inner structure of KExpressions, which consists of the three meta-models *KExpressions*, *KEffects* and *KExt*.

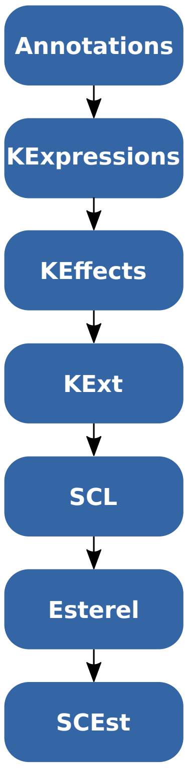


Figure 5.1. Hierarchical structure of the different meta-models.

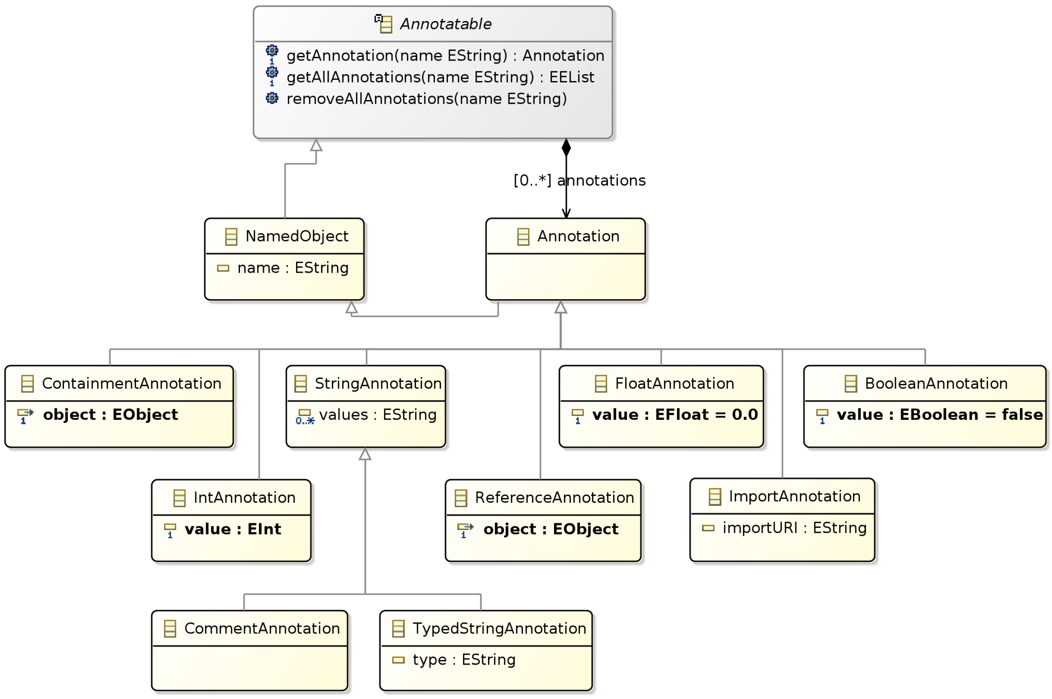


Figure 5.2. Class diagram of the annotations meta-model.

5.1 Underlying Models

The meta-models in this section were already implemented in KIELER and are just explained for a better understanding of the whole underlying structure.

5.1.1 Annotation Implementation

The base of all necessary plugins is the *Annotations* plugin. `Annotations` are implemented in KIELER in Xtext and the EMF framework. The `Annotations` meta-model, which is shown in Figure 5.2, includes the basic language feature of annotating an object to store extra information. The class diagram can be read as follows: While a square indicates a class, a black arrow indicates that the class at which the arrow starts has a list of objects of the class' type at which the arrow points, or just a single attribute of that class. The arrow has also a label which shows the list's or attribute's name and also an indication on how many objects can be added to the list, e.g., one to infinity `[1..*]` or zero to infinity `[0..*]`, or if an instance of the attribute must exist, e.g., the attribute may exist `[0..1]` or must exist `[1..1]`. A gray arrow with the not filled out arrow head indicates that the source class at which the arrow starts is derived from the target class. The derived class has all the attributes and methods of its parent class. In this example `TypedStringAnnotation` has also a list of values, which it has inherited from `StringAnnotation`. Also, it has a name because `StringAnnotation` inherited it from `Annotation`, which inherited it from `NamedObject`. The color of the `Annotable` square at the top indicates that the `Annotable` class is an abstract class, which means that it can not be instantiated. It specifies that all classes that are derived from it have the three methods mentioned in the square. After a colon the type of the attribute or list is indicated.

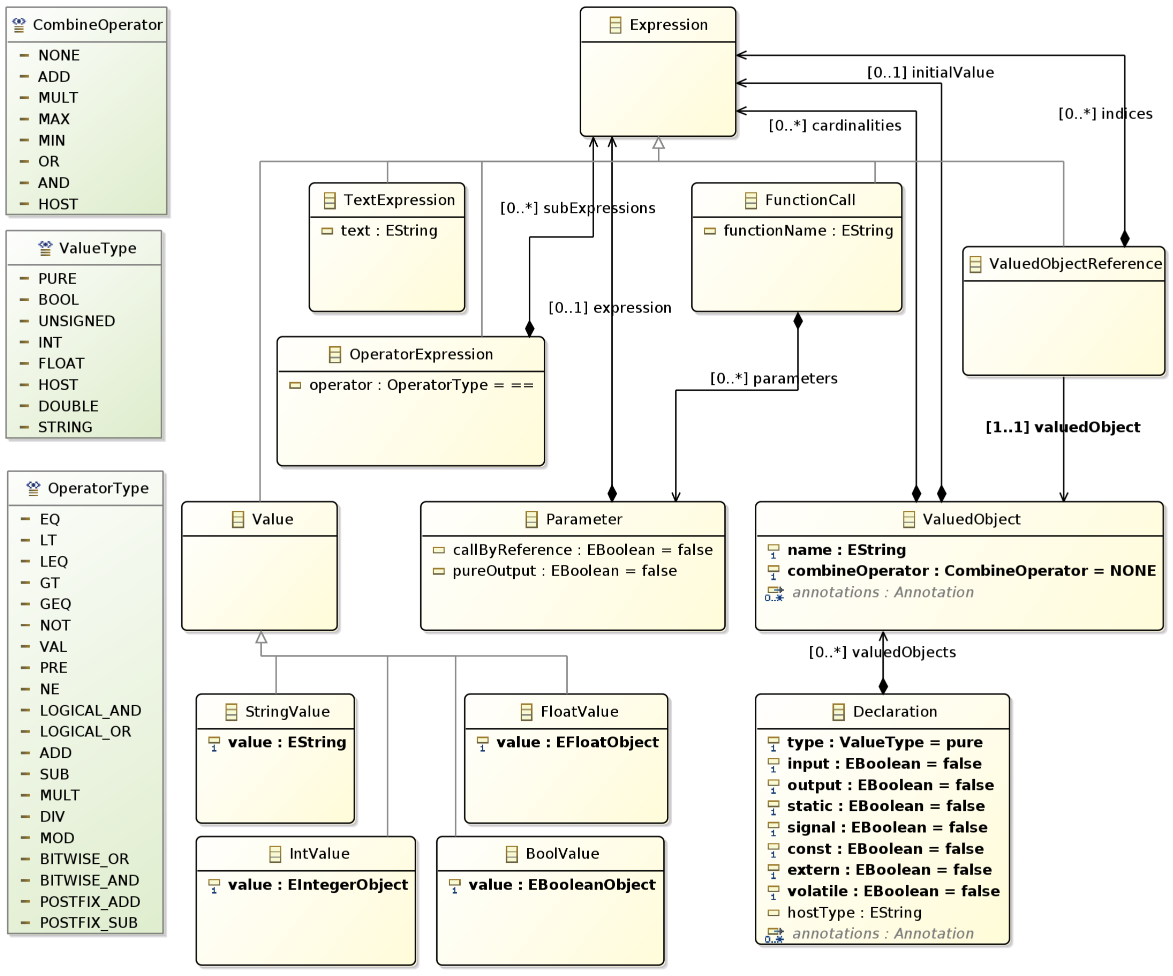


Figure 5.3. Class diagram of the KExpressions meta-model.

5.1.2 KExpression Implementation

The *KExpressions* meta-model, which is shown in Figure 5.3, expands the annotations meta-model in the way that it provides expressions like simple values (1, true, etc.), valued operator expressions (addition, subtraction, multiplication, etc.), or boolean operator expressions (logical or, logical and, not, etc.). The KExpressions are not purely used for SCL, Esterel and SCEst, they are also used for example for SCCharts. A `VlauedObjectReference` is an `Expression`, indicated by the arrow with the empty arrow head from `ValuedObjectReference` to `Expression` for example. It must contain a reference to a `ValuedObject`, indicated by the label `[1..1]` next to the arrow from `ValuedObjectReference` to `ValuedObject`. Also, an `OperatorExpression` is an `Expression`, has an `OperatorType`, which is an enumeration, displayed at the left in the green box, and contains a list `subExpression` which itself contains an arbitrary number from zero to infinity of objects of the type `Expression`, indicated by the arrow with the black diamond and the label `[0..*]`.

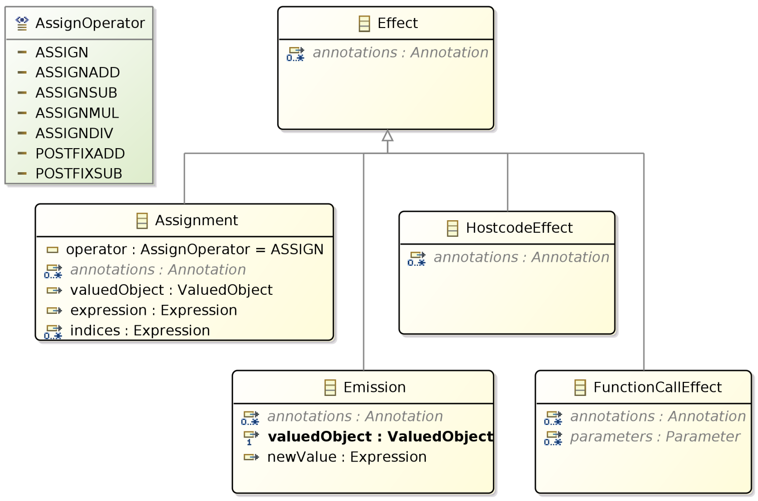

AssignOperator
ASSIGN
ASSIGNADD
ASSIGNSUB
ASSIGNMUL
ASSIGNDIV
POSTFIXADD
POSTFIXSUB
Effect
0..* annotations : Annotation
Assignment
operator : AssignOperator = ASSIGN
0..* annotations : Annotation
valuedObject : ValuedObject
expression : Expression
0..* indices : Expression
HostcodeEffect
0..* annotations : Annotation
Emission
0..* annotations : Annotation
1 valuedObject : ValuedObject
newValue : Expression
FunctionCallEffect
0..* annotations : Annotation
0..* parameters : Parameter

Figure 5.4. Class diagram of the KEffects meta-model.

```
1  Declaration returns kexpressions::Declaration:
2      (annotations+=Annotation)*
3      (const?='const')?
4      (extern?='extern')?
5      (volatile?='volatile')?
6      (input?='input')?
7      (output?='output')?
8      (static?='static')?
9      (((signal?='signal')? type = ValueType) | (signal?='signal'))
10     valuedObjects+=ValuedObject (',' valuedObjects+=ValuedObject)* ';';
```

Listing 5.1. Specification on how to build a Declaration, provided in KExt.

5.1.3 KEffects Implementation

The *KEffects* meta-model, derived from KExpressions and shown in Figure 5.4, provides a range of effects for deriving grammars. A KEffect is namely an assignment, a postfix effect, an emission, a host code effect or a function call.

5.1.4 KExt Implementation

KExt, derived from KEffects, provides a connection between KExpressions/KEffects and a grammar, which wants to use them, in the way that KExt offers a declaration mechanism, which is written in Xtext. It is shown in Listing 5.1. In lines 3 to 8 possible additional information on the declaration can be added by writing `const`, `input` or `output` for example. In line 10 variables as `ValuedObjects` are added to the declaration. An arbitrary number of variables, but at least one, have to be declared, separated by a comma.

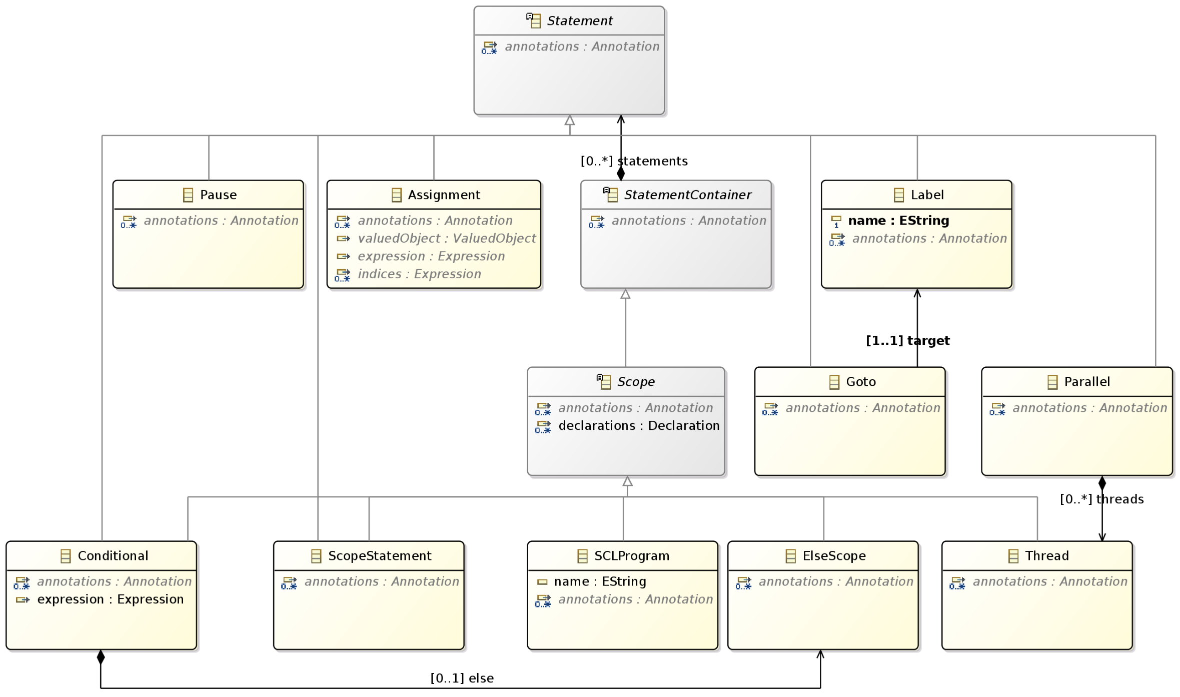


Figure 5.5. Class diagram of the SCL meta-model.

5.1.5 SCL Implementation

SCL is implemented in KIELER in Xtext and the EMF framework. Even though scoping, formatting and code validation is used in SCL, scoping and formatting are described in Section 5.4 for Esterel, while code validation is described in Section 5.4 for SCEst, since it is the same concept.

EMF Meta-Model of SCL

Since SCL is needed to transform an Esterel/SCEst program to target code with the already existing compilation chain from SCL over SCG to target code, the meta-model of SCL is displayed in this subsection. Figure 5.5 shows the meta-model of SCL. `SCLProgram` is the root object, which is visualized by no incoming arrows. It is a `Scope` which itself is a `StatementContainer`, which contains a list of zero to an arbitrary number of statements, indicated by the black diamond with its range `[0..*]`. A `Statement` is either an `InstructionStatement` or a `MetaStatement`. Figure 5.5 does not include every inheritance which can be seen with the *Assignment* statement, as it has all of its attributes in light grey, which indicates that they are all inherited, but only the `annotations` from `Statement` can be derived in this view. The `Assignment` statement is derived from the assignment of KEffects. Within this thesis just a few small adjustments were made to the SCL meta-model. For example, the `StatementContainer` was moved from the Esterel meta-model to the SCL meta-model, to be able to apply it to SCL statements also.

```
SCLProgram returns SCLProgram:
    (annotations += Annotation)*
    'module' name = ID ':'?
    (declarations+=Declaration)*
    ( (statements += InstructionStatement';') |
      (statements += MetaStatement)
    )*
    (statements += Statement)?;

Statement returns Statement:
    InstructionStatement | MetaStatement;

InstructionStatement returns Statement:
    Assignment | Conditional | Goto | Parallel | Pause;

MetaStatement returns Statement:
    Label | ScopeStatement;
```

Listing 5.2. The basic structure of an SCL program.

SCL Xtext Grammar

The grammar of SCL is described in Xtext, which is described in Section 2.1.2. To facilitate a better understanding of valid SCL programs and the basic structure of the SCL grammar implementation, an extract of the SCL grammar implementation is shown in Listing 5.2, including the root object of an SCL program, an `SCLProgram`, and the two different kind of statements, the `InstructionStatement` and the `MetaStatement`. An `SCLProgram` begins with optional annotations, stored in its list `annotations`, followed by the keyword `module` and its `ID`, which is stored in its attribute `name`. After an optional colon, `Declaration` objects, which are provided by KExt, can be added to its list `declarations`. Its last part is a sequence of statements which are separated by semicolons, unless its a `Label` or a `ScopeStatement`, and stored in its list `statements`. A `Statement` is either an `InstructionStatement` or a `MetaStatement`, it is just an abstract class with no attributes. It is the superclass of all statements. An `InstructionStatement` is either an `Assignment`, a `Conditional`, a `Goto`, a `Parallel` or a `Pause`. A `MetaStatement` is either a `Label` or a `ScopeStatement`.

Listing 5.3 shows another extraction of the SCL grammar, the `Conditional` statement. A `Conditional` begins, like the `SCLProgram`, with a list of annotations, followed by the `if` keyword, an `Expression` which is provided by KExpressions and stored in its attribute `expression`. The body of a `Conditional` starts with an optional `then` keyword, followed by the keyword `{`. It follows a list of declarations and thereafter a list of statements, similar to the `SCLProgram`. The body is closed by the keyword `}`. The last part is an optional `ElseScope`, which is stored in the attribute `else`. The `ElseScope` starts, again, with a list of annotations, followed by the keywords `else` and `{`. It follows a list of declarations and subsequently a list of statements. It is closed by the keyword `}`. Note that there is no `else if` in a `Conditional` statement in SCL. Therefore, `Conditionals` have to be nested when needed, which means in the `ElseScope` of a `Conditional` another `Conditional` has to be placed and when desired in the `ElseScope` of the second `Conditional` a third `Conditional` has to be placed. With this approach an arbitrary number of `else if` cases can be generated. As with the meta-model of SCL, the grammar of SCL was just barely modified during this thesis.

```
Conditional returns Conditional:
    (annotations += Annotation)*
    'if' expression = Expression
    (
        'then'? '{'
        (declarations+=Declaration)*
        ( (statements += InstructionStatement';') |
          (statements += MetaStatement)
        )*
         (statements += Statement)?
        '}'
    )
    (else = ElseScope)?;

ElseScope returns ElseScope:
    {ElseScope}
    (annotations += Annotation)*
    'else' '{'
    (declarations+=Declaration)*
    ( (statements += InstructionStatement';') |
      (statements += MetaStatement)
    )*
    (statements += Statement)?
    '}';
```

Listing 5.3. The implementation of the `Conditional` statement of SCL in Xtext.

5.2 Esterel Implementation

Esterel is implemented in KIELER in Xtext and the EMF framework. Esterel uses the implementation of KExpressions for its expressions, but since it needs a few extra options, the KExpressions are extended within the Esterel meta-model. For example, Esterel has a `SignalExpression` which does not include a standard `ValuedObjectReference` object but a `SignalReferenceExpr` object.

As described before, an implementation of Esterel in KIELER already existed. The meta-model adjustment, that the Esterel meta-model is derived from the SCL meta-model, was mentioned in Chapter 4. A main objective was the reduction of the number of objects, since it was inconvenient for a programmer to produce multiple instances of objects just for a single statement.

An example is shown in Listing 5.4. It displays the Xtext implementation of the previous Esterel `Module`, whereas Listing 5.5 displays the new implementation. When a programmer needed a new module, an instance of a `Module` had to be created. Additionally, an instance of `ModuleBody` and `EndModule` had to be created since they were mandatory for a valid Esterel `Module`, lines 1 and 2 in Listing 5.4. If, for example, a signal declaration was wanted, which usually is, also an instance of `ModuleInterface` was needed. Therefore, a programmer had to create at least 3 instances, but most of the time 4 instances, to be able to have a valid Esterel `Module`. The new implementation in Listing 5.5 from line 4 to 17 lets a programmer create just one instance of an Esterel `Module` to have a valid module. This implementation makes a module instantiation much easier for a programmer; therefore, the new approach should be superior to the previous one.

The same pattern, the minimization of rules, was used for all Esterel Xtext grammar rules. This lets a programmer write a more compact and comprehensible program.

```
1 TrapExpression returns kexpressions::Expression:
2     {TrapExpression} "??" trap=[kexpressions::ISignal|ID];
```

```
1 TrapExpression returns kexpressions::Expression:
2     {TrapExpression} "??" trap=[ISignal|ID];
```

Figure 5.6. Adjustment of the `TrapExpression` in the new Esterel grammar implementation.

In addition, as mentioned in Chapter 4, the KExpressions of Esterel were adjusted in the way that the expression rules in the Esterel grammar create KExpressions of which the SCL grammar is derived from. The different implementations of the `TrapExpression` in Figure 5.6 do not seem different, except that in line two `kexpressions::ISignal` in the first listing and `ISignal` in the second listing indicate that `ISignal` is now implemented in the Esterel Xtext grammar and not in an extern Xtext grammar as before. However, `kexpressions::` in line 1 of both listings reference different meta-models. In the previous grammar implementation, the KExpression just for Esterel were referenced, whereas in the new grammar implementation the KExpressions, shown in Section 5.1.2, are referenced.

Further, Figure 5.7 shows the adjustment of an `OrExpression`. As mentioned with `TrapExpression`, `kexpressions::` differ in the referenced meta-model. The main difference here is best displayed with the following example. If the expression `A or B or C or D` was written in the Esterel editor, it led to an instance of the `OrExpression` for `C` and `D`. Additionally, this instance was part of a second instance of the `OrExpression` together with `B`. Lastly a third instance of the `OrExpression` was created, including the second instance and `A`. The new implementation leads to the intended creation of an `OrExpression`, just one instance, having references to `A`, `B`, `C` and `D` in its list `subExpressions`.

Lastly, the possibility to annotate each object was added within the new Esterel grammar implementation, which can be seen in line 5 of Listing 5.5 for example.

5.2.1 EMF Meta-Model of Esterel

To compare the size of the SCL meta-model to the Esterel meta-model, Figure 5.8 shows the meta-model of Esterel in the tree editor view, since the diagram view is not comprehensible. This figure gives just an overview of the different objects included in the Esterel meta-model. The arrow after the object's name indicates that the object is derived from the object after the arrow. For example, at the top left, a `Module` is derived from `StatementContainer`, indicating that it has a list of statements. At the bottom right, the before mentioned additions to the KExpressions meta-model can be seen, indicated by deriving the KExpressions rules `ValuedObjectReference` and `Expression`. `Program` is the root object. It has zero to an arbitrary number of modules, eight declarations lists, explained in Section 5.2.2, and an attribute `name`.

5.2.2 Esterel Xtext Grammar

The grammar of Esterel is described in Xtext, which is described in Section 2.1.2. For a better understanding on how a valid Esterel program can be written, the overall pattern of the Esterel Xtext grammar and the interaction between the Esterel, SCL and KExpression grammar implementations, the

```
BooleanExpression returns kexpressions::Expression:
    OrExpression;

OrExpression returns kexpressions::Expression:
    AndExpression
    (   {OperatorExpression.subExpressions+=current}
        operator=Esterel_OrOperator
        subExpressions+=AndExpression
    )*;
```

```
BooleanExpression returns kexpressions::Expression:
    OrExpression;

OrExpression returns kexpressions::Expression:
    AndExpression
    (   {kexpressions::OperatorExpression.subExpressions+=current}
        (operator=Esterel_OrOperator subExpressions+=AndExpression)+
    )?;
```

Figure 5.7. Adjustment of the `OrExpression` in the new Esterel grammar implementation.

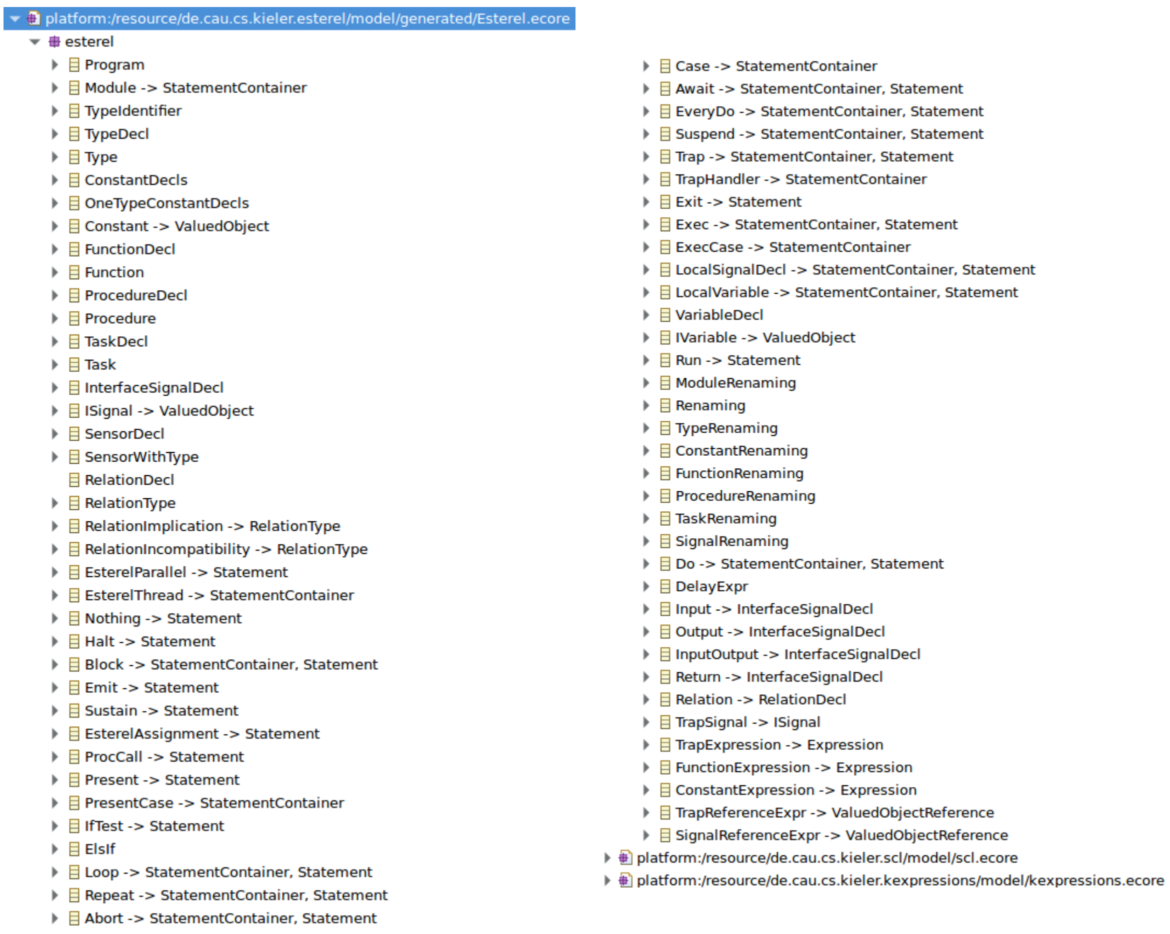
```
Module:
  "module" name=ID ":" (interface=ModuleInterface)? body=ModuleBody end=EndModule;

EndModule:
  "end" "module"
  | ".";

ModuleBody:
  statements+=Statement;

ModuleInterface:
  (intSignalDecls+=InterfaceSignalDecl
  | intTypeDecls+=TypeDecl
  | intSensorDecls+=SensorDecl
  | intConstantDecls+=ConstantDecls
  | intRelationDecls+=RelationDecl
  | intTaskDecls+=TaskDecl
  | intFunctionDecls+=FunctionDecl
  | intProcedureDecls+=ProcedureDecl)+;
```

Listing 5.4. The previous Xtext implementation of an Esterel `Module`.



```
platform:/resource/de.cau.cs.kieler.esterel/model/generated/Esterel.ecore
  esterel
    Program
    Module -> StatementContainer
    TypeIdentifier
    TypeDecl
    Type
    ConstantDecls
    OneTypeConstantDecls
    Constant -> ValuedObject
    FunctionDecl
    Function
    ProcedureDecl
    Procedure
    TaskDecl
    Task
    InterfaceSignalDecl
    ISignal -> ValuedObject
    SensorDecl
    SensorWithType
    RelationDecl
    RelationType
    RelationImplication -> RelationType
    RelationIncompatibility -> RelationType
    EsterelParallel -> Statement
    EsterelThread -> StatementContainer
    Nothing -> Statement
    Halt -> Statement
    Block -> StatementContainer, Statement
    Emit -> Statement
    Sustain -> Statement
    EsterelAssignment -> Statement
    ProcCall -> Statement
    Present -> Statement
    PresentCase -> StatementContainer
    IfTest -> Statement
    ElsIf
    Loop -> StatementContainer, Statement
    Repeat -> StatementContainer, Statement
    Abort -> StatementContainer, Statement
    Case -> StatementContainer
    Await -> StatementContainer, Statement
    EveryDo -> StatementContainer, Statement
    Suspend -> StatementContainer, Statement
    Trap -> StatementContainer, Statement
    TrapHandler -> StatementContainer
    Exit -> Statement
    Exec -> StatementContainer, Statement
    ExecCase -> StatementContainer
    LocalSignalDecl -> StatementContainer, Statement
    LocalVariable -> StatementContainer, Statement
    VariableDecl
    IVariable -> ValuedObject
    Run -> Statement
    ModuleRenaming
    Renaming
    TypeRenaming
    ConstantRenaming
    FunctionRenaming
    ProcedureRenaming
    TaskRenaming
    SignalRenaming
    Do -> StatementContainer, Statement
    DelayExpr
    Input -> InterfaceSignalDecl
    Output -> InterfaceSignalDecl
    InputOutput -> InterfaceSignalDecl
    Return -> InterfaceSignalDecl
    Relation -> RelationDecl
    TrapSignal -> ISignal
    TrapExpression -> Expression
    FunctionExpression -> Expression
    ConstantExpression -> Expression
    TrapReferenceExpr -> ValuedObjectReference
    SignalReferenceExpr -> ValuedObjectReference
platform:/resource/de.cau.cs.kieler.scl/model/scl.ecore
platform:/resource/de.cau.cs.kieler.kexpressions/model/kexpressions.ecore
```

Figure 5.8. Tree editor view of the Esterel meta-model.

rules for a `Program`, a `Module`, an `EsterelStatement` and an `AtomicStatement`, as well as an `IfTest` are described in the following.

Listing 5.5 visualizes a part of the Esterel grammar. It shows the root object of an Esterel program, a `Program`. It consists of a list of zero to an arbitrary number of modules. A module, `Module`, starts with zero to any number of annotations, which are stored in the list `annotations`, followed by the keyword `module` and the module's `ID`, which is stored in its attribute `name`. After the colon keyword, an arbitrary number of declarations follow. There are eight kinds of declarations and therefore eight lists for different declarations, a list for signals, for types, for sensors, for constants, for relations, for tasks, for functions and for procedures. After the declarations follows a sequence of statements, which are stored in the list `statements`. The end of a statement is indicated by a semicolon, except for the last statement of the sequence, which can but does not have to end with a semicolon. A `Module` statement ends with the keyword `.` or the keywords `end module`. An `EsterelStatement` is either an `AtomicStatement` or an `EsterelParallel`. As indicated by `returns scl::Statement`, an `EsterelStatement` is a descendant of the abstract `Statement` of SCL, which is useful for SCEst. The distinction had to be made since the definition of `EsterelParallel`, a sequence of statements divided by the keyword `||`, combined with the used approach of a sequence of statements, would end in a left recursive grammar, which is not allowed in

```
Program hidden(SL_COMMENT, ML_COMMENT, WS):
    (modules+=Module)*;

Module:
    (annotations += Annotation)*
    "module" name=ID ":"
    (     intSignalDecls+=InterfaceSignalDecl
        | intTypeDecls+=TypeDecl
        | intSensorDecls+=SensorDecl
        | intConstantDecls+=ConstantDecls
        | intRelationDecls+=RelationDecl
        | intTaskDecls+=TaskDecl
        | intFunctionDecls+=FunctionDecl
        | intProcedureDecls+=ProcedureDecl
    )*
    ( (statements+=EsterelStatement ";")* statements+=EsterelStatement? )
    ("end" "module" | ".");

EsterelStatement returns scl::Statement:
    EsterelParallel | AtomicStatement;

AtomicStatement returns scl::Statement:
    Abort | EsterelAssignment | Await | Block | ProcCall | Do | Emit | EveryDo |
    Exit | Exec | Halt | IfTest | LocalSignalDecl | Loop | Nothing | Pause | Present |
    Repeat | Run | Suspend | Sustain | Trap | LocalVariable ;
```

Listing 5.5. The basic structure of an Esterel program.

Xtext.

Listing 5.6 shows the `IfTest` statement of Esterel. It starts with an optional list of annotations, followed by the keyword `if` and thereafter an `Expression`, which is provided by KExpressions and stored in its attribute `expression`. The following body is optional. It starts with a list of annotations, followed by the keyword `then` and ends with a list of statements. If there is more than one `Expression` which needs to be tested, at this point the `IfTest` statement provides a list of `ElsIf` objects, which are defined like the `IfTest` up to this point, except for the keyword `if`, which is replaced by the keyword `elsif`. After the `ElsIf` objects the optional else case follows, which starts again with a list of annotations, followed by the keyword `else` and ends with a list of statements. The `IfTest` ends with the keyword `end` or the keywords `end if`.

5.3 SCEst Implementation

SCEst extends Esterel in a few small ways, as described previously in Section 1.2.2. However, the implementation of SCEst differs from the formal definition of SCEst by also including SCL statements as shown in the following.

```
IfTest:
    (annotations += Annotation)*
    "if"
    expression=Expression
    (
        (thenAnnotations += Annotation)*
        "then"
        ( (thenStatements+=EsterelStatement ";")* thenStatements+=EsterelStatement? )
    )?
    (elseif+=ElsIf)*
    (
            (elseAnnotations += Annotation)*
            "else"
            ( (elseStatements+=EsterelStatement ";")* elseStatements+=EsterelStatement? )
    )?
    "end" "if"?;

ElsIf:
    (annotations += Annotation)*
    "elsif"
    expression=Expression
    (
        "then"
        ( (thenStatements+=EsterelStatement ";")* thenStatements+=EsterelStatement? )
    )?;
```

Listing 5.6. The implementation of the `IfTest` statement of Esterel in Xtext.

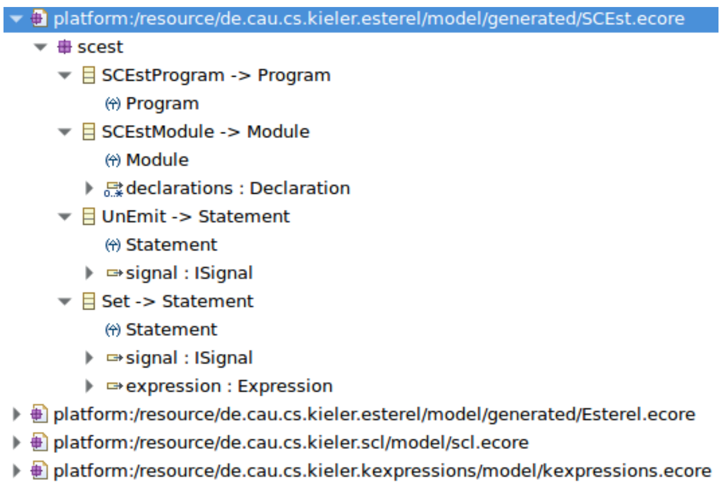
5.3.1 EMF Meta-Model of SCEst

For a comparison between the SCEst meta-model, its different objects and its size, to the Esterel meta-model, Figure 5.9 shows the meta-model of SCEst. `SCEstProgram` is the root object. In contrast to the meta-model of Esterel, the meta-model of SCEst is very small. That is, because SCEst expands Esterel only in two new statements, the `UnEmit` and the `Set` statement, and the new definition of the `SCEstProgram` and the `SCEstModule`. A `SCEstProgram` has zero to an arbitrary number of modules, which is declared in the Esterel meta-model, since the `SCEstProgram` is derived from the Esterel `Program`, indicated by the arrow after `SCEstProgram` to `Program`. A `SCEstModule` is derived from `Module` but in addition, it contains a list `declarations` to store declarations of type `Declaration`, shown in Section 5.1.4. An `UnEmit` is of type `Statement` and contains a reference to an `ISignal`. The `Set` statement is the same, except that it also has a reference to an expression.

5.3.2 SCEst Xtext Grammar

The grammar of SCEst is described in Xtext, which is described in Section 2.1.2. As with the Esterel grammar, in the following an extract of the SCEst grammar implementation is described to give a better understanding on how a valid SCEst program can be written, the overall pattern of the SCEst Xtext grammar, and the interaction between the SCEst, Esterel, SCL and KExpressions grammar implementations.

Listing 5.7 shows an extract of the SCEst grammar. The root object of a SCEst program, a `SCEstProgram`



```
platform:/resource/de.cau.cs.kieler.esterel/model/generated/SCEst.ecore
  scest
    SCEstProgram -> Program
      Program
    SCEstModule -> Module
      Module
      declarations : Declaration
    UnEmit -> Statement
      Statement
      signal : ISignal
    Set -> Statement
      Statement
      signal : ISignal
      expression : Expression
platform:/resource/de.cau.cs.kieler.esterel/model/generated/Esterel.ecore
platform:/resource/de.cau.cs.kieler.scl/model/scl.ecore
platform:/resource/de.cau.cs.kieler.kexpressions/model/kexpressions.ecore
```

Figure 5.9. Tree editor view of the SCEst meta-model.

consists of a list of zero to an arbitrary number of modules. It also returns a `Program` which is implemented in Esterel, defined by `ProgramInterface`. A `SCEstModule` returns a `Module`, which is implemented in Esterel, defined by `ModuleInterface`. A `SCEstModule` consists of an arbitrary number of annotations, followed by the keyword `module`, an `ID` which is saved in its attribute `name`, and a colon. The body of a `SCEstModule` is either like the body of a `Module`, which means it starts with an arbitrary number of declarations - the eight different declaration types are for signals, types, sensors, constants, relations, tasks, functions and procedures - which are saved in its eight different declaration lists, followed by an arbitrary number of statements, which are saved in the list `statements` and separated by a semicolon except for `MetaStatements`, and ends with the keyword `.` or the keywords `end module`, or the body of a `SCEstModule` is like the body of an `SCLProgram`. The latter means that it starts with an arbitrary number of declarations, which are saved in its list `declarations`, followed by the keyword `{`, an arbitrary number of statements, stored and written like in the previous case, and ends with the keyword `}`.

Listing 5.8 shows the definition of a `SCEstStatement` and a `SCEstAtomicStatement` as well as the `StatementContainerInterface`. Latter returns a `StatementContainer`, which is implemented in SCL, and it is either a `SCEstModule`, a `Conditional`, an `ElseScope`, a `Thread` or a `ScopeStatement`. Except for the first statement, the statements are just redefinitions of the SCL statements in the way that they allow `SCEstStatements` in their list of statements. A `SCEstStatement` is either an `EsterelParallel` or a `SCEstAtomicStatement` and returns a `Statement`, which means that it extends `Statement`, implemented in SCL. A `SCEstAtomicStatement` also returns a `Statement` and it differs from an `AtomicStatement`, which is implemented in Esterel, in the way that it also is an `UnEmit`, a `Set`, a `Goto`, an `Assignment`, a `Parallel` and a `Conditional` statement. The first two are new statements, which are part of the definition of SCEst, the last four are statements from SCL. Since the implemented SCEst language integrates all statements of SCL, it differs from the formal definition of SCEst [RSM+15]. But it simplifies the use for programmers. It is also necessary for the step by step transformation, because it allows to have one model on which the transformations take place.

```
SCEstProgram hidden(SL_COMMENT, ML_COMMENT, WS):
    (modules+=SCEstModule)*;

ProgramInterface returns esterel::Program:
    SCEstProgram;

ModuleInterface returns esterel::Module:
    SCEstModule;

SCEstModule:
    {SCEstModule}
    (annotations += Annotation)*
    "module" name=ID ":"
    (
        (     intSignalDecls+=InterfaceSignalDecl
            | intTypeDecls+=TypeDecl
            | intSensorDecls+=SensorDecl
            | intConstantDecls+=ConstantDecls
            | intRelationDecls+=RelationDecl
            | intTaskDecls+=TaskDecl
            | intFunctionDecls+=FunctionDecl
            | intProcedureDecls+=ProcedureDecl
        )*
        ( ( statements+=SCEstStatement ";" | statements+=MetaStatement)*
          statements+=SCEstStatement? )
        ("end" "module" | ".")

    |

        (declarations+=Declaration)*
        '{'
        ( ( statements+=SCEstStatement ";" | statements+=MetaStatement)*
          statements+=SCEstStatement? )
        '}'
    );
```

Listing 5.7. An extract of the SCEst grammar in Xtext which shows the root object.

```
StatementContainerInterface returns scl::StatementContainer:
    SCEstModule | Conditional | ElseScope | Thread | ScopeStatement;

SCEstStatement returns scl::Statement:
    EsterelParallel | SCEstAtomicStatement;

SCEstAtomicStatement returns scl::Statement:
    Abort | EsterelAssignment | Await | Block | ProcCall | Do | Emit | EveryDo |
    Exit | Exec | Halt | IfTest | LocalSignalDecl | Loop | Nothing | Pause | Present |
    Repeat | Run | Suspend | Sustain | Trap | LocalVariable | Goto | UnEmit | Set |
    Assignment | Parallel | Conditional ;
```

Listing 5.8. A second extract of the SCEst grammar in Xtext which shows the declaration of statements in SCEst.

```
for (Keyword at : f.findKeywords("@")) {
  c.setNoSpace().after(at);
}

// Block statement
c.setLinewrap().before(f.getBlockAccess().getLeftSquareBracketKeyword_2());
c.setLinewrap().before(f.getBlockAccess().getRightSquareBracketKeyword_4());
c.setIndentation( f.getBlockAccess().getLeftSquareBracketKeyword_2(),
          f.getBlockAccess().getRightSquareBracketKeyword_4());
```

Listing 5.9. Two example parts of Esterel's formatter.

5.4 Assisting Features

In this section three assisting features for formatting, scoping and code validation are being discussed.

5.4.1 Formatting

For comprehensibility reasons it is necessary to format a generated code. Without formatting, a generated code would just be a one line string without extra blank characters. For this reason Xtext provides a formatter which extends the `AbstractDeclarativeFormatter` class. An instance of the formatter, which has the information on how the code should be formatted, together with an instance of the `GrammarAccess` class, which includes methods to find specific elements such as keywords of the language, are used to correctly format the generated code. In the first part of Listing 5.9, it shows how annotations are formatted. First all `@` keywords of a given code are collected by the use of the method `findKeyword` of the `GrammarAccess` class, followed by setting no space after each `@` keyword with the `setNoSpace` method of the formatter class. The second part of Listing 5.9 shows the formatting of a `Block` statement, which begins with the `[` keyword. Therefore, a line wrap should be set before this keyword, which is done by calling the method `setLinewrap`. Also it is possible to set an indentation with the method `setIndentation` which requires a start and an end point. The first listing in Figure 5.10 shows an Esterel program which is valid but not formatted. After using the formatter, the code looks like the one shown in the second listing in Figure 5.10.

5.4.2 Scoping

Since Esterel needs some kind of scoping, which determines for example which signals can be referenced from a specific point, it comes in handy that Xtext provides a scoping API. An example is shown in Listing 5.10. Even though output signal `A` is emitted in line 4, `B` will be absent, since the present test for `A` in line 7 references the first enclosing signal of that name, which is the local signal declaration in this case. If the `Present` statement tested for the signal `C`, `B` would be emitted, since the local signal declaration does not include a signal with the name `C` and therefore the next enclosing scope would be tested for a declaration for this name.

The `EsterelScopeProvider` extends the `AbstractDeclarativeScopeProvider`. The previous Esterel meta-model implementation had a scope provider which was extended during this thesis to facilitate a correct behavior after the modification of the meta-model. Listing 5.11 shows that an `Emit` statement triggers the method `getAllSignals` of the `EsterelScopeProviderUtil` class to determine which signal should be referenced. Listing 5.12 shows the definition of this method. First, in line 3, the method

```
module test: input A;[ if A then pause end if] end module
```

```
module test:
input A;
[
  if A then
    pause
  end if
]
end module
```

Figure 5.10. Example program before and after formatting.

```
module scoping:
output A, B, C;
[
  emit A;
  emit C;
  signal A in
    present A then
      emit B
    end present
  end signal
]
end module
```

Listing 5.10. An example Esterel program for scoping where the output signal `A` should be shadowed by the local signal declaration of `A`.

`getLocalSignals` will be triggered which collects all enclosing local signals. In line 13 it is checked if `Module` is reached or if there is another enclosing statement to check for local signal declarations. Inside the loop, in line 15, it is checked if the enclosing statement is a local signal declaration and if so, all its signals are added to the list `scopeElems`. Therefore, the first element in the list `scopeElems` will be the closest occurrence of a local signal. After the search for local signals, a list of all signals from the modules interface are added to `scopElems` by calling `getAllSignals`. Since the first element of the list with the right name will be used for the `Emit` statement, a local signal will be prioritized over a signal in the module's interface.

5.4.3 Code Validation

Xtext automatically generates an abstract class providing an interface for the `SCEstValidator` class, which is a collection of methods which check for errors in a given SCEst program inside of the SCEst editor to indicate or highlight errors or warnings. Debugging can be simplified that way since the user sees errors and warnings directly and therefore is able to write a valid program in the first place.

```java
public class EsterelScopeProvider extends AbstractDeclarativeScopeProvider {
    public IScope scope_Emit_signal(final Emit context, final EReference ref) {
        return new SimpleScope(getAllSignals(context));
    }
}
```

Listing 5.11. A few lines of the `EsterelScopeProvider`.

```java
public final class EsterelScopeProviderUtil {
  public static List<IEObjectDescription> getAllSignals(final EObject context) {
        List<IEObjectDescription> scopeElems = getLocalSignals(context);
        scopeElems.addAll(getAllElements(context, COLLECT_SIGNALS));
        return scopeElems;
    }

    public static List<IEObjectDescription> getLocalSignals(final EObject context) {
        ArrayList<IEObjectDescription> scopeElems = new ArrayList<IEObjectDescription>();

        EObject parent = context.eContainer();
        // Go up in the Structure until Module/MainModule
        while (!(parent instanceof Module)) {
            // Get the local signals into the scope
            if (parent instanceof LocalSignalDecl) {
                EList<ISignal> signals = ((LocalSignalDecl) parent).getSignals();
                for (ISignal s : signals) {
                    scopeElems.add(new EObjectDescription(QualifiedName.create(s.getName()),
                    s, getEmptyMap(String.class)));
                }
            }
            parent = parent.eContainer();
        }

        return scopeElems;
    }
}
```

Listing 5.12. The method `getAllSignals` of the `EsterelScopeProvider`.

Listing 5.13 shows two example methods of the `SCEstValidator` class. The `@Check` annotation indicates that it should be checked if an object of the method's parameter type exists and if so, the method should be initiated on this object. A `@Check` annotation can have three different types, indicating how often a check should be activated. First, `@Check(CheckType.FAST)` indicates that the check is initiated every time the file is modified. Second, `@Check(CheckType.NORMAL)` just runs the check when the file is being saved. Lastly, `@Check(CheckType.EXPENSIVE)` is only run when the user explicitly starts a validation check. The first method `emitSignal` is invoked for every `Emit` statement and checks in line 4 if the emitted signal is a pure signal. If so, the emit can not be a valued emit. But if it is, an error will be displayed, lines 6 and 7. The first argument of the error message is the message itself, while the second argument defines the faulty statement. By setting the third argument to `null` and the last to `-1`, the whole statement will be highlighted. If the signal is a valued signal, the emit must be a valued emit; otherwise, an error will be displayed, lines 12 and 13. An example for an error is shown in Figure 5.11. The error is marked in red and when the cursor is on top of the error mark, the black box with a message appears.

The second method `expressionEmit` is also invoked for every `Emit` statement. If it is a valued emit, line 20, it is checked if the expression fits the type of the valued signal. If not, a warning will be displayed, line 22 or 26 and 27. An example for a warning is shown in Figure 5.11. The warning is marked in yellow and when the cursor is on top of the warning, the black box with a message appears.

5.5 Incremental Compilation Implementation

The implementation of the incremental compilation of SCEst is integrated in the Esterel plugin with the name `de.cau.cs.kieler.esterel`. The current approach is to combine related implementations in one plugin.

As explained in Chapter 1, KiCo and the SLIC approach are used for that task. All transformations extend the `AbstractExpansionTransformation` class for the interaction with KiCo[1] as shown in line 1 of Listing 5.14, a sample of the `LoopTransformation` class. While the method `getId` provides the ID of a specific transformation, `getName` provides the name of a specific transformation, in this example lines 2 to 4 and lines 5 to 7 respectively. The method `getExpandsFeatureId` returns the ID of the corresponding feature of a specific transformation, lines 8 to 10. There are two methods that specify which transformations should be done before the given transformation, the `getNotHandlesFeatureIds` method, and which objects might be created during the given transformation and therefore which transformations, the corresponding transformations to the created objects, should be done after the given transformation, the `getProducesFeatureIds` method. In this example the `LoopTransformation` can produce `Abort` and `Halt` statements, lines 11 to 13, and therefore should be scheduled before those transformations. Additionally, it must be scheduled after the `Initialization` and `Run` transformation, indicated in lines 14 to 16. In general, a specific transformation is called by the method `transform(SCEstProgram prog)`, line 17, while the return value is a `SCEstProgram`, except for the last transformation which gets an instance of a SCEst meta-model, a completely transformed `SCEstProgram` which only consists of SCL statements, and returns the given program as an instance of the SCL meta-model.

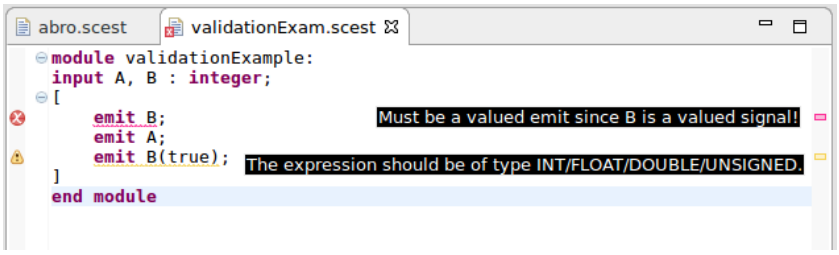
In `de.cau.cs.kieler.esterel.scest.extensions` the class `SCEstExtension` is located, which is written in Xtend. It is a collection of helper methods for the single transformations and some variables and a map for the communication between transformations, since one transformation for example can transform signals to new variables, but another transformation is used to transform a statement which references the old signal. Therefore, the second transformation needs a mapping from the old signals

[1]https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Kieler+Compiler

```
class SCEstValidator extends SCEstJavaValidator{
  @Check
  def void emitSignal(Emit emit) {
    if (emit.signal.type.isPure) {
      if (emit.expression != null) {
        error(emit.signal.name +
        " is not a valued signal!", emit, null, -1)
      }
    }
    else {
      if (emit.expression == null) {
        error("Must be a valued emit since " +
          emit.signal.name + " is a valued signal!", emit, null, -1)
      }
    }
  }

  @Check
  def void expressionEmit(Emit emit) {
    if (emit.expression != null) {
      if (emit.signal.type.isBool && !emit.expression.isBoolExpr) {
        warning("The expression should be of type BOOL.", null)
      }
      else if ( emit.signal.type.isCalculationType
            && !emit.expression.isCalculationExpr) {
        warning("The expression should be of
              type INT/FLOAT/DOUBLE/UNSIGNED.", null)
      }
    }
  }
}
```

Listing 5.13. An extract of the `SCEstValidator` class.



```
module validationExample:
input A, B : integer;
[
    emit B;            Must be a valued emit since B is a valued signal!
    emit A;
    emit B(true);  The expression should be of type INT/FLOAT/DOUBLE/UNSIGNED.
]
end module
```

Figure 5.11. An example for code validation.

```
class LoopTransformation extends AbstractExpansionTransformation implements Traceable{
    override getId() {
        return SCEstTransformation::LOOP_ID
    }
    override getName() {
        return SCEstTransformation::LOOP_NAME
    }
    override getExpandsFeatureId() {
        return SCEstFeature::LOOP_ID
    }
    override getProducesFeatureIds() {
        return Sets.newHashSet(SCEstTransformation::ABORT_ID, SCEstTransformation::HALT_ID)
    }
    override getNotHandlesFeatureIds() {
        return Sets.newHashSet(SCEstTransformation::INITIALIZATION_ID, SCEstTransformation::RUN_ID)
    }
    def SCEstProgram transform(SCEstProgram prog) {
        prog.modules.forEach [ m | transformStatements(m.statements)]
        return prog
    }
}
```

Listing 5.14. An extract of the `LoopTransformation` class.

to the new corresponding variables.

The single M2M transformations are located in `de.cau.cs.kieler.esterel.scest.transformations` and are written in Xtend, described in Section 2.1.3. Figure 5.12 is a screenshot of the KiCo View. It shows all transformations, which can be selected by the user, and their dependencies. Transformations can be grouped to make the KiCo View clearer, as it is done in the previously mentioned figure.

There are 30 transformation classes plus one transformation, which transforms a given instance of an Esterel meta-model to an instance of a SCEst meta-model (which in concrete terms means that the `Program` will be transformed to a `SCEstProgram` and every `Module` will be transformed to a `SCEstModule`), and which is located in `de.cau.cs.kieler.esterel.transformations`. The latter is only included in the Esterel editor in KIELER and done before all other transformations.

In the following, the extension class as well as the standard transformations, which are used in every transformation, are being outlined. Since it would not be beneficial to expound on all 30 transformations of specific SCEst statements, this section explains just a few single transformations as an example to show the general implementation pattern.

5.5.1 SCEst Extension

The `SCEstExtension` class provides a collection of methods for the single transformation classes; therefore, methods are not written just for one specific transformation in its transformation class, but rather they are written just once to be used by multiple transformation classes. Also the code of the transformations is easier to comprehend this way.

Listing 5.15 shows nine counting variables. They are used to create unique names for newly created variables of different types. It could have been just one counter, but this way the numbers and therefore the new names do not have as many digits. Following the nine counting variables, there are three

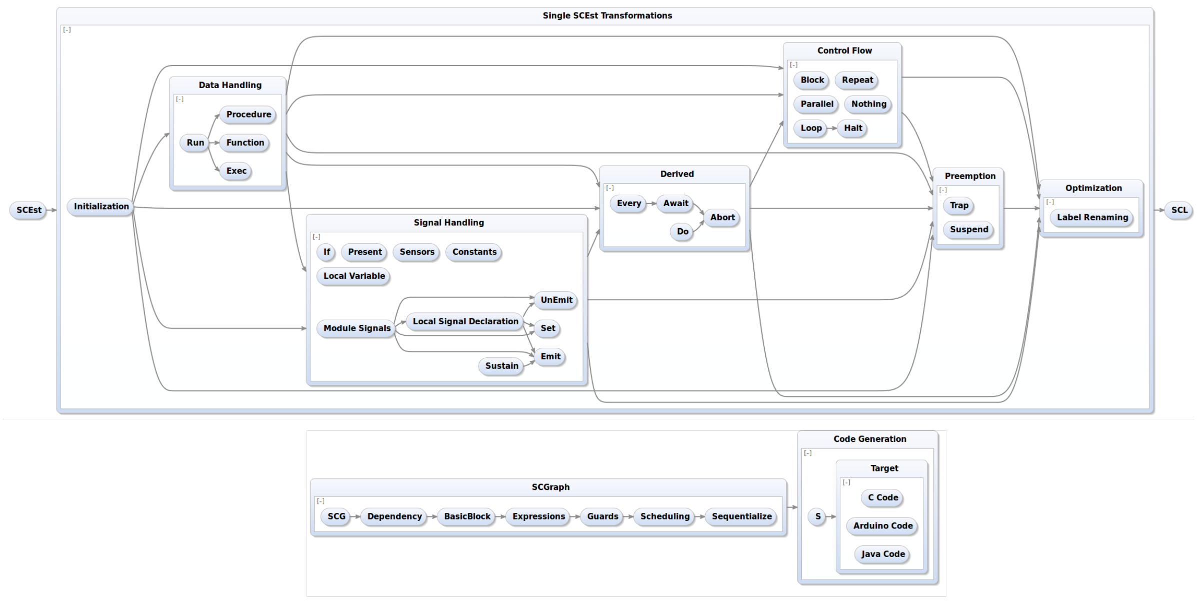


Figure 5.12. The KiCo View with all single transformations for a SCEst program.

```
var static labelSuffix = 0;
var static constantSuffix = 0;
var static variableSuffix = 0;
var static flagSuffix = 0;
var static abortFlagSuffix = 0;
var static depthFlagSuffix = 0;
var static trapSuffix = 0;
var static signalSuffix = 0;
var static moduleSuffix = 0;

final private static String generatedAnnotation = "depth"

final private static String interfaceScope = "IScope"

final private static String generatedModule = "generatedModuleForRun"

// for valued singals: signal S will be transformed to s, s_set, s_cur, s_val =>
// new NewSignals(s, s_set, s_cur, s_val)
var static HashMap<ISignal, NewSignals> newSignals = new HashMap<ISignal, NewSignals>()
```

Listing 5.15. Variables and a map of the `SCEstExtension` class.

String variables that set the name of specific annotations, which are used for communication between single transformations. Lastly, the map `newSignals` includes a mapping from the old signals of Esterel to the new variables which will represent the old signals in the newly created SCL program, again, for communication between single transformations. The class `NewSignals` is just a collection of four variables, because a signal will be transformed into one or four variables, described in Section 4.2.

Listing 5.16 shows the method `createConditional` of the `SCEstExtension` class. The method is called

```
def createConditional(Expression expr) {
  SclFactory::eINSTANCE.createConditional => [
      it.expression = EcoreUtil.copy(expr)
  ]
}
```

Listing 5.16. The method `createConditional` of the `SCEstExtension` class.

```
def Statement transformStatement(Statement statement) {
  if (statement instanceof IfTest) {
      var ifTest = statement as IfTest
      var conditional = SclFactory::eINSTANCE.createConditional => [
                      it.expression = EcoreUtil.copy(ifTest.expression)
                  ]
      ...
  }
}
```

Listing 5.17. Example code which does not have access to the extension class.

with an `Expression` as parameter. The `SclFactory` instantiates a `Conditional` with its expression being a copy of the given `Expression`. Therefore, the code in Listing 5.17, which would have to be written without the helper method, can be written as the code in Listing 5.18. In this easy example it would still be comprehensible, but nonetheless the written code is cleaner and easier to write when the extension method is called multiple times. Note that in Xtend its possible to write in line 4 in Listing 5.18 `ifTest.expression.createConditional` instead of `createConditional(ifTest.expression)`, although it is not defined in the `Expression` class. It is called *extension methods* in Xtend.

Another example method is the `checkGoto` method, shown in Listing 5.19, which goes over a list of statements and checks for every statement if it is a `Goto` statement by calling the method `checkGoto` in line 3. If so, the method `findClosestLabel(Label label, Statement statement)` is called in line 10, which returns either the given label or a label which is at the end of the enclosing thread when the given label is outside of its enclosing thread.

```
1 def Statement transformStatement(Statement statement) {
2   if (statement instanceof IfTest) {
3       var ifTest = statement as IfTest
4       var conditional = createConditional(ifTest.expression)
5       ...
6   }
7 }
```

Listing 5.18. Example code which has access to the extension class.

```
def void checkGotos(EList<Statement> statements) {
  statements?.forEach [ s |
      s.checkGoto
  ]
}

def void checkGoto(Statement statement) {
  if (statement instanceof Goto) {
      var goto = (statement as Goto)
      goto.target = findClosestLabel(goto.target, statement)
  }
  else if (statement instanceof StatementContainer) {
  ...
  }
}
```

Listing 5.19. The method `checkGoto` of the `SCEstExtension` class.

5.5.2 Initialization Transformation

The `InitializationTransformation` class is called in the beginning of every transformation chain. It makes sure that all counting variables are reset to 0 and the signals map is cleared, shown in lines 2 to 5 in Listing 5.20. Afterwards, every `SCEstModule` that was not generated for a `Run` statement is being transformed in the following way by calling the method `transformStatements` in line 9.

For some transformations it is necessary to know if another statement has priority. An example is being shown in Figure 5.13. Both, the `Suspend` and the `Abort` statement, will transform the `Pause` statement, but the `Abort` statement has priority. Since they can change places and therefore the `Suspend` statement will be the outermost statement which has priority, there can not be a predefined order of the transformations. Thus, the priority of each statement has to be saved in the meta-model. This is accomplished by adding an annotations to each statement, which saves the depth of the given statement. The first level, the statements in the list `statements` of a module, starts with a depth of 1. Is at the first level a statement which includes other statements, they will have the depth 2. When a statement of the latter has a statement list, the statements in this list will have the depth 3 and so on. The result can be seen in Figure 5.13a. If the `Abort` transformation transforms the `Pause` statement first, the `Suspend` transformation will recognize that the conditional, which has the priority of the `Suspend` statement and which it wants to place after the `Pause` statement, has a higher depth and therefore a lower priority than the conditional behind the `Pause` statement. Thus, the conditional of the transformed `Suspend` statement will be placed after the conditional of the transformed `Abort` statement. Which means in this example, which is shown in Figure 5.13b, after the transformation, the program executes in its intended way. Hence, `emit X` will be executed when `A` is absent. On the other hand, when `A` is present, `emit X` will not be executed. Figure 5.14 shows how the different transformations are selected in the KiCo View.

During the previously mentioned procedure, a `Run` statement will be transformed in the way that its module will be copied, because multiple `Run` statements can reference the same module and all of them could be transformed differently since the transformation of a module depends on the context of the `Run` statement. Listing 5.21 shows the transformation of a `Run` statement. In line 7 it is checked if the given module is already generated. If so, nothing has to be copied. If not, the module will be copied, line 8, the copy of the module will be set to be the module of the `Run` statement, line 9, the new

```
def SCEstProgram transform(SCEstProgram prog) {
  resetLabelSuffix
  ...
  resetModuleSuffix
  clearNewSignalsMap
  for (var i=0; i<prog.modules.length; i++) {
      var m = prog.modules.get(i)
      if (!m.annotations.isGeneratedModule) {
      transformStatements(m.statements, 1)
      }
  }
  return prog
}
```

Listing 5.20. Beginning of the transformation of the `InitializationTransformation` class.

module will get a unique name, line 10, and the module will get an annotation which indicates that it has been generated, line 11. Afterwards `Renaming` has to be adjusted, line 12, and finally the generated module has to be added to the `SCEstProgram`, the root object, in line 13.

5.5.3 SCL Transformation

The `SCLTransformation` class gets an instance of a SCEst meta-model, a `SCEstProgram`, removes the old local signals and also the annotations which were used for the transformations and are no longer of use, and returns an instance of the SCL meta-model, an `SCLProgram`.

5.5.4 Label Renaming Transformation

The `OptimizationLabelRenamingTransformation` class holds the last transformation before the SCL transformation in a transformation chain. Because of multiple transformations, the first occurring label in the generated program code is not always the one with the lowest number. Note that the generated labels are named as followed: `l<counter>`. Examples are `l1`, `l2` or `l10`. Thus, the code is not as comprehensible as it could be. So this transformation makes sure that a label in an arbitrary line has a higher number in its name than a label in a previous line.

5.5.5 Block Transformation

The `BlockTransformation` class provides a transformation for an Esterel `Block`. For a given `SCEstProgram` every `SCEstModule` is transformed in the form of going through all statements of this module and checking if a given statement is a `Block` statement. Listing 5.22 shows how a Block statement is being transformed. In line 2 it is checked if the given statement is a `Block` statement. If so, in line 3 a new `ScopeStatement` of SCL is created. After the statements of the `Block` statement have been transformed, the statements will be moved to the new `ScopeStatement`, line 4.

5.5.6 SCEst Transformation

The `SCEstTransform` class, located in `de.cau.cs.kieler.esterel.transformations`, provides a transformation for an Esterel `Program` and its Esterel `Modules` to a `SCEstProgram` and `SCEstModules`. This

```
module abortSuspend:
input A,B;
output X;
abort
  suspend
    pause;
  when B;
  emit X;
when A;
end module
```

Initialization

```
module abortSuspend_t1:
input A,B;
output X;
@depth 1
abort
  @depth 2
  suspend
    pause;
  when B;
  emit X;
when A;
end module
```

(a) The example after `InitializationTransformation` is used.

Abort

```
module abortSuspend_t2:
input A,B;
output X;
@depth 2
suspend
  pause;
  @depth 1
  if A {
    goto l1;
  };
when B;
emit X;
l1:
end module
```

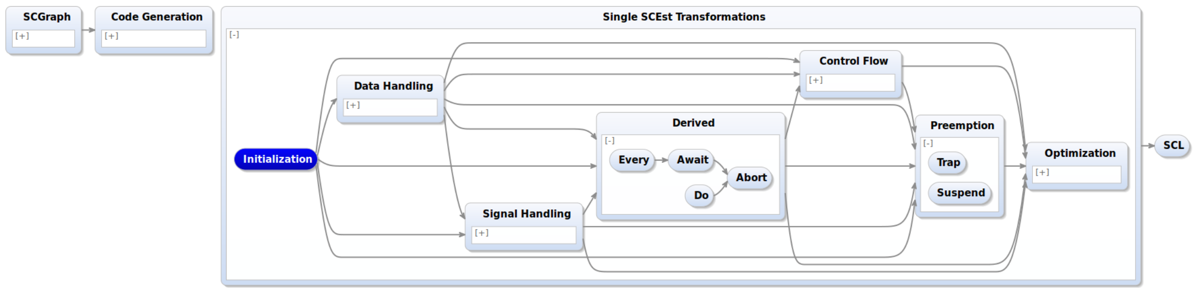
Suspend

```
module abortSuspend_t3:
input A,B;
output X;
l2:
pause;
@depth 1
if A {
  goto l1;
};
@depth 2
if B {
  goto l2;
}
emit X;
l1:
end module
```

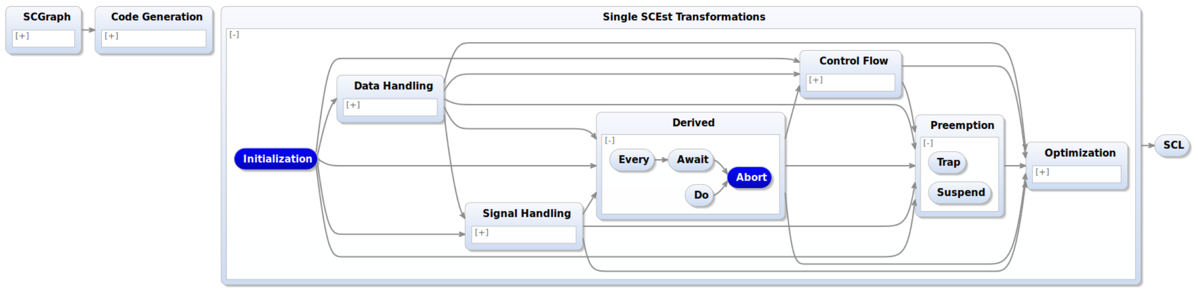
(b) The example after `AbortTransformation` and `SuspendTransformation` is used respectively.

Figure 5.13. Transformation of an example Esterel program with an `Abort` and `Suspend` statement.

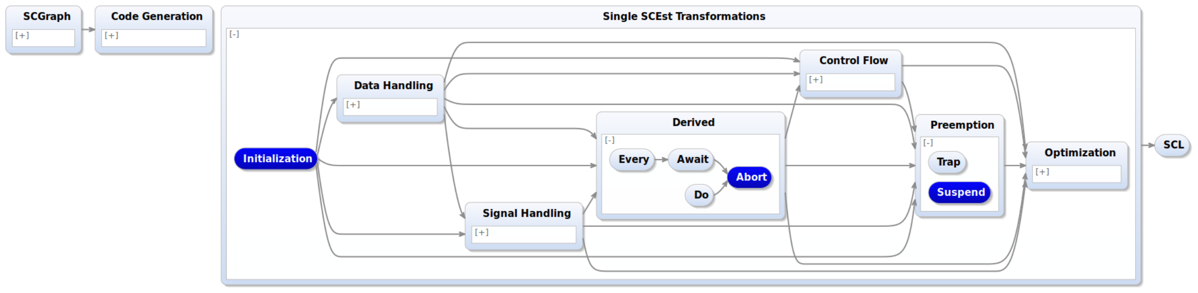
transformation is only called in the Esterel editor and not in the SCEst editor. It is the first transformation, even before the `InitializationTransformation`. Therefore, it is ensured that the following transformations get a `SCEstProgram` to do their transformations on. Since a `SCEstProgram` extends an Esterel `Program` and a `SCEstModule` extends an Esterel `Module`, the respective parts just have to be copied. This means for an Esterel `Module` that its statements and its interface declarations will be copied to a newly created `SCEstModule`. Since the `InitializationTransformation` will copy modules in case of a `Run` statement, `SCEstTransform` copies a module of a `Run` statement and marks it as generated. Therefore, during the following execution of `InitializationTransformation` it will be recognized that the module is already generated.



(a) The selection of `InitializationTransformation`.



(b) The selection of `InitializationTransformation` and `AbortTransformation`.



(c) The selection of `InitializationTransformation`, `AbortTransformation` and also `SuspendTransformation`.

Figure 5.14. The KiCo View for the previous example of a step by step transformation while the Esterel program includes an `Abort` and a `Suspend` statement.

```
def Statement transformStatement(Statement statement, int depth) {
  ...
  else if (statement instanceof Run) {
      // create a copy of the referenced module
      // so that the original module will not be modified
      // and update the renamings
      if (!statement.module.module.annotations.isGeneratedModule) {
      var moduleCopy = EcoreUtil.copy(statement.module.module)
      statement.module.module = moduleCopy
      moduleCopy.name = moduleCopy.name.createNewUniqueModuleName
      moduleCopy.annotations.add(createModuleAnnotation)
      statement.transformRenamingsAfterModuleCopy
      scestProgram.modules.add(moduleCopy)
      }
      transformStatements(statement.module.module.statements, depth+1)
  }
}
```

Listing 5.21. The `Run` statement transformation of the `InitializationTransformation` class.

```
def Statement transformStatement(Statement statement) {
  if (statement instanceof Block) {
      return createScopeStatement(null) => [
    statements.add(transformStatements((statement as Block).statements))]
  }
  ...
}
```

Listing 5.22. Extract of the `BlockTransformation` class.

Chapter 6

Evaluation

This chapter covers an evaluation of the incremental compilation of SCEst designed and implemented during this thesis. A step by step transformation of the `ABRO` example from a SCEst program to SCL is shown and described to demonstrate how each intermediate step is displayed. Hence, it is easier for the user to understand how the resulting SCL program was generated. Also, the number of statements of example SCEst programs is compared to the number of statements of the respectively generated SCL programs for this thesis' approach and for Rathlev's approach. Additionally, two problems which arose during the design and implementation of the incremental compilation are being discussed.

6.1 ABRO Step by Step Compilation

An example step by step transformation of `ABRO` is shown in Listing 6.1. Listing 6.2 to Listing 6.10 is the generated code of the compiler for every step of the compilation chain. All single transformation rules which are used in this section are described in Section 4.2. Their order is displayed in Figure 6.1, as the compilation starts with an initialization and ends with the `SCL` transformation. In the following, the more complex transformations will be explained. Note that every time a user transforms `ABRO`, a different order of the single transformations might be used as far as the produce and not-handled-by dependencies allow it, as described in Section 1.4.

The `Initialization` transformation adds annotations to the model, as shown in Listing 6.2, to save extra information which is later used by single transformations to determine the placement of their conditionals after/before pauses, as described in Section 5.5.2.

Before an `Emit` statement can be transformed, new representations of the signals have to be created. Therefore, the transformation of the signals is the second, as shown in Listing 6.3. A `ScopeStatement` is generated to represent a module of SCEst in SCL. The transformation marks the generated scope with the annotation `@IScope`. Hence, if the transformation of sensors for example is next, the already existing scope can be identified as the scope for the interface declarations of the module. Thus, it is also used for the new declarations for the sensors. Note that the transformation of the signals generates an extra thread which sets the output signals to false at the beginning of every tick. Since the whole program is transformed to SCL, an SCL parallel statement is generated as shown in lines 11, 20 and 39 of Listing 6.3 and not a parallel statement of Esterel. Also, the old signals stay part of the generated program, as mentioned in Section 4.2.2. Otherwise, this step would need to include the transformation of other statements like the `Emit` statement, since an `Emit` statement needs to reference a signal. However, this would lead to a transformation of many kinds of statements in one step.

The `SCL` transformation, as described in Section 5.5.3, transforms the instance of the SCEst meta-model to an instance of the SCL meta-model, which leads to Listing 6.11.

It is shown for this example, that the transformation is done as intended. Each intermediate result could be serialized and therefore shown to the user. Looking at this transformation step by step makes it easier for a user to understand how the resulting SCL program emerged, in contrast to the previous structural approach, where the user just got the final SCL program. Also, possible errors in the whole

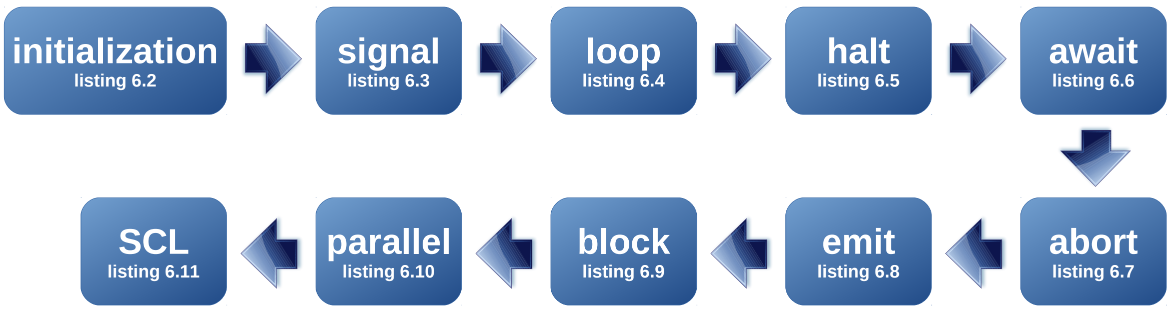


Figure 6.1. Illustration of the incremental steps of the compilation of `ABRO`.

transformation can be narrowed down relatively fast by inspecting the intermediate steps. In addition, the user can see which transformations are simple and which are more complex. For that reason, the user might be able to reduce the number of statements in the resulting SCL program by choosing Esterel statements in the source program that do not translate into as many SCL statements.

```
module ABRO:
input A, B, R;
output O;
loop
  abort
    [
      await A
      ||
      await B
    ];
    emit O;
    halt;
  when R;
end loop
end module
```

Listing 6.1. The `ABRO` example ...

initialization ⇨

```
module ABRO_01:
input A,B,R;
output O;
@depth 1
loop
  @depth 2
  abort
    [
      @depth 5
      await A;
      ||
      @depth 5
      await B;
    ];
    emit O;
    halt;
  when R;
end loop;
end module
```

Listing 6.2. ... after the `initialization` transformation, ...

signal

```
module ABRO_02:
input A_sig, B_sig, R_sig;
output O_sig;
@IScope
{
  input bool A = false;
  input bool B = false;
  input bool R = false;
  output bool O = false;
  bool term1 = false;
  fork
    {
      _l1:
      O = false;
      if !term1 {
        pause;
        goto _l1;
      };
    }
  par
    {
      @depth 1
      loop
        @depth 2
        abort
          [
            @depth 5
            await A_sig;
            ||
            @depth 5
            await B_sig;
          ];
          emit O_sig;
          halt;
        when R_sig;
      end loop;
      term1 = true;
    }
  join;
  }
end module
```

Listing 6.3. ... after the signal transformation, ...

loop

```
module ABRO_03:
input A_sig, B_sig, R_sig;
output O_sig;
@IScope
{
  input bool A = false;
  input bool B = false;
  input bool R = false;
  output bool O = false;
  bool term1 = false;
  fork
    {
      _l1:
      O = false;
      if !term1 {
        pause;
        goto _l1;
      };
    }
  par
    {
      _l2:
      @depth 2
      abort
        [
          @depth 5
          await A_sig;
          ||
          @depth 5
          await B_sig;
        ];
        emit O_sig;
        halt;
      when R_sig;
      goto _l2;
      term1 = true;
    }
  join;
}
end module
```

Listing 6.4. ... after the loop transformation, ...

halt

```
module ABRO_04:
input A_sig, B_sig, R_sig;
output O_sig;
@IScope
{
  input bool A = false;
  input bool B = false;
  input bool R = false;
  output bool O = false;
  bool term1 = false;
  fork
    {
      _l1:
      O = false;
      if !term1 {
        pause;
        goto _l1;
      };
    }
  par
    {
      _l2:
      @depth 2
      abort
        [
          @depth 5
          await A_sig;
          ||
          @depth 5
          await B_sig;
        ];
        emit O_sig;
        _l3:
        pause;
        goto _l3;
      when R_sig;
      goto _l2;
      term1 = true;
    }
  join;
}
end module
```

Listing 6.5. ... after the `halt` transformation, ...

await

```
module ABRO_05:
input A_sig, B_sig, R_sig;
output O_sig;
@IScope
{
  input bool A = false;
  input bool B = false;
  input bool R = false;
  output bool O = false;
  bool term1 = false;
  fork
    {
      _l1:
      O = false;
      if !term1 {
        pause;
        goto _l1;
      };
    }
  par
    {
      _l2:
      @depth 2
      abort
        [
          _l4:
          pause;
          if !A {
            goto _l4;
          };
          ||
          _l5:
          pause;
          if !B {
            goto _l5;
          };
        ];
        emit O_sig;
        _l3:
        pause;
        goto _l3;
      when R_sig;
      goto _l2;
      term1 = true;
    }
  join;
}
end module
```

Listing 6.6. ... after the `await` transformation, ...

```
module ABRO_06:
input A_sig, B_sig, R_sig;
output O_sig;
@IScope
{
  input bool A = false;
  input bool B = false;
  input bool R = false;
  output bool O = false;
  bool term1 = false;
  fork
    {
      _l1:
      O = false;
      if !term1 {
        pause;
        goto _l1;
      };
    }
  par
    {
      _l4:
      {
        bool abort1 = false;
        [
          _l2:
          pause;
          @depth 2
          if R {
            abort1 = true;
            goto _l7;
          };
          if !A {
            goto _l2;
          };
          _l7:
          ||
          _l3:
          pause;
          @depth 2
          if R {
            abort1 = true;
            goto _l8;
          };
          if !B {
            goto _l3;
          };
          _l8:
          if abort1 {
            goto _l6;
          };
        ];
        emit O_sig;
        _l5:
        pause;
        @depth 2
        if R {
          abort1 = true;
          goto _l6;
        };
        goto _l5;
        _l6:
      }
      goto _l4;
      term1 = true;
    }
  join;
}
end module
```

Listing 6.7. ... after the abort transformation, ...

```
module ABRO_07:
input A_sig, B_sig, R_sig;
output O_sig;
@IScope
{
 input bool A = false;
 input bool B = false;
 input bool R = false;
 output bool O = false;
 bool term1 = false;
 fork
  {
   _l1:
   O = false;
   if !term1 {
    pause;
    goto _l1;
   };
  }
 par
  {
   _l4:
   {
    bool abort1 = false;
    [
     _l2:
     pause;
     @depth 2
     if R {
      abort1 = true;
      goto _l7;
     };
     if !A {
      goto _l2;
     };
     _l7:
     ||
     _l3:
     pause;
     @depth 2
     if R {
      abort1 = true;
      goto _l8;
     };
     if !B {
      goto _l3;
     };
     _l8:
     if abort1 {
      goto _l6;
     };
    ];
    O |= true;
    _l5:
    pause;
    @depth 2
    if R {
     abort1 = true;
     goto _l6;
    };
    goto _l5;
    _l6:
   }
   goto _l4;
   term1 = true;
  }
 join;
}
end module
```

Listing 6.8. ... after the emit transformation, ...

block

```
module ABRO_08:
input A_sig, B_sig, R_sig;
output O_sig;
@IScope
{
  input bool A = false;
  input bool B = false;
  input bool R = false;
  output bool O = false;
  bool term1 = false;
  fork
    {
      _l1:
      O = false;
      if !term1 {
        pause;
        goto _l1;
      };
    }
  par
    {
      _l4:
      {
        bool abort1 = false;
        {
          _l2:
          pause;
          @depth 2
          if R {
            abort1 = true;
            goto _l7;
          };
          if !A {
            goto _l2;
          };
          _l7:
          ||
          _l3:
          pause;
          @depth 2
          if R {
            abort1 = true;
            goto _l8;
          };
          if !B {
            goto _l3;
          };
          _l8:
          if abort1 {
            goto _l6;
          };
        }
        O |= true;
        _l5:
        pause;
        @depth 2
        if R {
          abort1 = true;
          goto _l6;
        };
        goto _l5;
        _l6:
      }
      goto _l4;
      term1 = true;
    }
  join;
}
end module
```

Listing 6.9. ... after the `block` transformation, ...

parallel

```
module ABRO_09:
input A_sig, B_sig, R_sig;
output O_sig;
@IScope
{
  input bool A = false;
  input bool B = false;
  input bool R = false;
  output bool O = false;
  bool term1 = false;
  fork
    {
      _l1:
      O = false;
      if !term1 {
        pause;
        goto _l1;
      };
    }
  par
    {
      _l4:
      {
        bool abort1 = false;
        {
          fork
            {
              _l2:
              pause;
              @depth 2
              if R {
                abort1 = true;
                goto _l7;
              };
              if !A {
                goto _l2;
              };
              _l7:
            }
          par
            {
              _l3:
              pause;
              @depth 2
              if R {
                abort1 = true;
                goto _l8;
              };
              if !B {
                goto _l3;
              };
              _l8:
            }
          join;
          if abort1 {
            goto _l6;
          };
        }
        O |= true;
        _l5:
        pause;
        @depth 2
        if R {
          abort1 = true;
          goto _l6;
        };
        goto _l5;
        _l6:
      }
      goto _l4;
      term1 = true;
    }
  join;
}
end module
```

Listing 6.10. ... after the `parallel` transformation, ...

SCL

```
module ABRO
{
  input bool A = false;
  input bool B = false;
  input bool R = false;
  output bool O = false;
  bool term1 = false;
  fork
    {
      l1:
      O = false;
      if !term1 {
        pause;
        goto l1;
      };
    }
  par
    {
      l2:
      {
        bool abort1 = false;
        {
          fork
            {
              l3:
              pause;
              if R {
                abort1 = true;
                goto l4;
              };
              if ! A {
                goto l3;
              };
              l4:
            }
          par
            {
              l5:
              pause;
              if R {
                abort1 = true;
                goto l6;
              };
              if ! B {
                goto l5;
              };
              l6:
            }
          join;
          if abort1 {
            goto l8;
          };
        }
        O |= true;
        l7:
        pause;
        if R {
          abort1 = true;
          goto l8;
        };
        goto l7;
        l8:
      }
      goto l2;
      term1 = true;
    }
  join;
}
```

Listing 6.11. ... after the `labelrenaming` and `SCL` transformation.

6.2 Size Comparison to the Approach of Rathlev

In this section the sizes of six test Esterel/SCEst programs and their representing SCL programs are being discussed. The `ABRO` example is one of those.

For each test SCEst program the number of statements was counted by a function in the class `SCEstExtension`, which goes through a list of `EObjects` and counts the number of `Statements`. After the number was counted, the respective program was being transformed with the incremental compilation implemented during this thesis to an SCL program. Afterwards the number of statements of the resulting SCL program was counted. By *statement* an SCL `Statement` is meant, as SCEst statements are derived of the SCL `Statement`, as described in Section 5.3.2.

	Name	SCEst		SCL	
		this thesis	Rathlev	this thesis	Rathlev
1	test-pause1	1	2	1	8
2	test-suspend9	4	4	22	25
3	test-abro	6	7	38	38
4	test-trap12	8	10	22	25
5	test-trap15	12	15	43	47
6	test-weakabort2	33	38	146	216

Table 6.1. Six test programs and their number of statements in SCEst and the respective generated representation in SCL for both, the incremental compilation of this thesis and Rathlev's approach [Rat15].

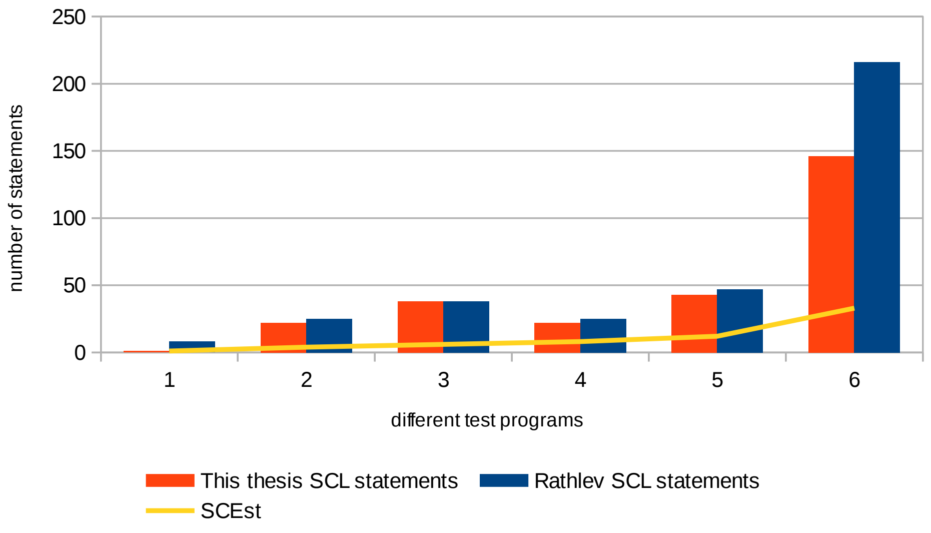


Figure 6.2. Illustration of the data of Table 6.1.

Table 6.1 shows the six test cases, which are additionally visualized in Figure 6.2. These are chosen as they were also used by Rathlev [Rat15].

Note that Rathlev's test included a few additional tests, which were affected by the problem of nested aborts and suspends, which is being discussed in the next section, or had many nested statements, which leads to the following. As explained in Section 2.1.2, the parser for the new Esterel Xtext grammar is auto-generated. As it proves to be the case, it can not handle deep nested statements in combination with Esterel's `EsterelParallel` statement for this new grammar because the parallel statement does not start with a keyword. Therefore, the parser does not recognize a parallel statement from the beginning. Just as the `||` keyword is reached, it realizes that all previous statements should have been in a thread. It results in a freeze of the Esterel/SCEst editor. Presumably, with the combination of SCL and Esterel/SCEst statements in this one grammar, the parser can not handle the amount of possibilities. In retrospect, this might have been already indicated by the long time needed for the generation of the Esterel/SCEst parser. Therefore, the implemented Xtext grammars for Esterel and SCEst needed to be improved regarding parser performance.

As part of the integration of this thesis' work into the KIELER project, the meta-model and the Xtext grammar of Esterel have already been modified by A. Schulz-Rosengarten. Hence, the parser does not have the before mentioned problem anymore. This was achieved by a less strict meta-model and a modification of the Xtext grammar; however, postprocessing had to be added for that reason. Also, the implemented transformations of this thesis must be adjusted to these modifications in the future.

Column three and four of Table 6.1 show the number of statements in the Esterel/SCEst source program for this thesis' approach and also for the one by Rathlev. They differ because the underlying meta-model was changed, as described in Section 4.1. In the previous meta-model every statement container had just a field to include a single statement. If multiple statements were wanted in a statement container, the statement `Sequence` was used, as it was the only statement which could include more than one single statement. For example, if just a single pause was wanted in a `Block` statement, it could be included in the `Block` statement directly. However, if, for example, an emit and a pause were wanted, the `Block` statement would include a `Sequence` statement and the `Sequence` statement then would include the emit and pause. Since `Sequence` was a statement, the total number of statements is higher for the previous meta-model.

Column five and six show the number of statements in the resulting SCL program after this incremental compilation was used and also for the structural compilation by Rathlev. As expected, the number of resulting SCL statements for test two to five does just barely differ as the same transformation rules were used. Since the resulting SCL programs of Rathlev's tests were not included in his thesis, it is not clear why `test-pause1` were transformed to eight statements during his structural compilation as it should include just one single pause. Also, why the transformation of `test-weakabort2` led to 216 statements instead of the 146 statements generated by the incremental compilation of this thesis can not be determined for certain. One possibility could be, that those are the numbers before his optimizations, which were partly directly included in the incremental compilation of this thesis. Nevertheless, a graph in his thesis shows that the optimized SCL code for `test-weakabort2` still has at least 200 statements. The meta-model of SCL has admittedly changed, but after comparing both structures, it should not result in a difference for the counting of the statements.

In summary, it can be stated that the implementation of the incremental compilation of this thesis does not produce more SCL statements than Rathlev's structural compilation approach. This was expected because the same transformation rules were used. However, in some cases this thesis' approach produces considerably less SCL statements, which can partly be attributed to optimizations that were directly included in the transformations of this thesis.

6.3 Self-Contained Intermediate Models

As described in Section 5.5.1, a mapping from the old signals to the new representing variables are saved in the map `newSignals` of the `SCEstExtension` class in the implementation written within this thesis. This extra information violates a part of the intermediate model condition of SLIC. While each intermediate model is a valid model, it is not self-contained because of the extra information. Again, this is specific to the implementation within this thesis. The mapping from the old signals to the new variables could also be saved in a `ReferenceAnnotation`. Thus, each new variable could be annotated with a reference to the old signal. Additionally, this could be circumvented, if during the transformation of the signals, all statements which reference one of the signals are being transformed as well, as described in Section 4.2.2. As mentioned there, this would come at the cost of not being able to create a set of features, each representing a statement of SCEst, and transforming each feature just once during the compilation, which is also part of the SLIC approach.

6.4 Nested Aborts and Suspends

With the incremental compilation a problem arose, which did not occur with the structural approach by Rathlev. The transformation of an `Abort` or `Suspend` statement adds conditionals after/before pauses in the scope of their body. Since `Suspend` and `Abort` statements can both generate pauses in specific cases, it is not achievable with the SLIC approach to transform the generated pauses in a correct way, as each transformation of `aborts` or `suspends` can only be done once (the single-pass condition of SLIC, Section 1.4). The following example visualizes the problem.

Listing 6.12 shows an `Abort` statement in a `Suspend` statement. The suspension is triggered if signal `A` is present and the abort is initiated if signal `B` is present for the second time (`abort when 2 B`). Since the `Suspend` statement surrounds the `Abort` statement, it has priority. This means, is signal `A` present and simultaneously signal `B` is present for the second time, the execution of the body of the `Suspend` statement will be suspended and the presence state of signal `B` will not count.

```
module SuspendAbort:
input A,B;
suspend
  abort
    pause;
  when 2 B;
when A;
end module
```

Listing 6.12. An `Abort` statement is inside a `Suspend` statement.

```
module AbortSuspend:
input A,B;
abort
  suspend
    pause;
  when 2 B;
when A;
end module
```

Listing 6.13. A `Suspend` statement is inside an `Abort` statement.

Listing 6.13 shows a `Suspend` statement in an `Abort` statement. The abort is initiated if signal `A` is present and the suspension is triggered if signal `B` is present for the second time (`suspend when 2 B`). Since the `Abort` statement surrounds the `Suspend` statement, it has priority. This means, is signal `A` present and simultaneously signal `B` is present for the second time, the execution of the statements inside the `Abort` statement's body will be aborted and the counter for the presence state of signal `B` does not matter anymore anyway.

Both listings show nested `Abort` and `Suspend` statements and indicate, that the order in which they are nested can be different and therefore a produce or not-handled-by dependency, which would set a fix order of the transformations for every future transformation, can not resolve the problem. This is visualized in Figure 6.3. Only if the `Abort` transformation is done before the `Suspend` transformation, Listing 6.12 is transformed in the intended way. On the other side, only if the `Suspend` transformation is done before the `Abort` transformation, Listing 6.13 is transformed in the intended way.

Listing 6.14 shows the correct transformation of Listing 6.12, while Listing 6.15 shows an incorrect transformation as explained in the following. The difference lies in the first thread, as the second threads are the same. The two threads are created by the `Abort` statement, since it needs a thread to count how many times signal `B` has been present. But if the `Suspend` statement is transformed first, the threads and the pause inside the first thread do not exist at that time. Therefore, the needed conditional, which is shown in lines 16 to 18 in Listing 6.14, can not be placed, as shown in Listing 6.15. Since the `Suspend` statement surrounds the `Abort` statement, its expression has priority over the one of the `Abort` statement. Therefore, both threads have to be suspended if signal `A` is present, otherwise the first thread would still increment the variable `v1`, which counts how many times the signal `B` has been present. In the next tick an incorrect abort could be activated that way.

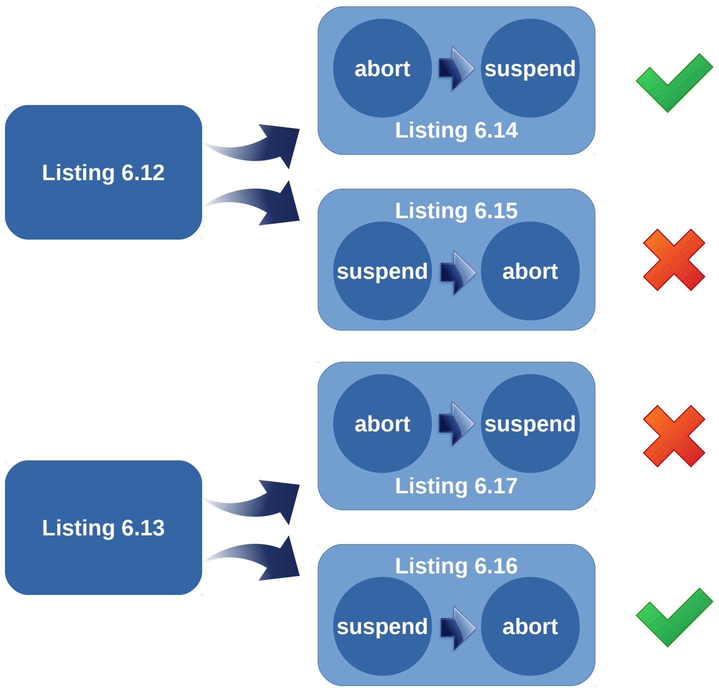


Figure 6.3. Illustration of the correct and incorrect order of the `Abort` and `Suspend` transformation for Listing 6.12 and Listing 6.13.

Listing 6.16 shows the correct transformation of Listing 6.13, while Listing 6.17 shows an incorrect transformation, as explained in the following. The second threads are the same. Note that the first four missing lines of Listing 6.16 are the same as the first four lines of Listing 6.17. The transformation of the `Suspend` statement leads to the creation of two threads, since it has to count the times signal `B` is present. This is done by incrementing the variable `v1` (line 25 in Listing 6.16 and line 21 in Listing 6.17). However, is the `Abort` statement being transformed first, the threads and especially the pause in the first thread do not exist at that moment. Therefore, the conditional in lines 17 to 20 in Listing 6.16 for the abort will not be placed, as shown in Listing 6.17. This might end in thread 2 being aborted before the termination flag `f1` is set to true, line 37 in Listing 6.17, and therefore thread 1 would run indefinitely, because it terminates only if `f1` is true, lines 13 and 14 in Listing 6.17. Thus, the execution of the whole program would not continue as intended. Additionally, the conditional after the `fork par join` construct, lines 45 to 47 in Listing 6.16, can not be placed, as shown in Listing 6.17. This could end in the execution of possible statements after the `Suspend` statement, which should not be executed since an abort was activated by signal `A` being present.

As shown, a correct transformation of nested `Abort` and `Suspend` statements can not be guaranteed by this implementation of the incremental compilation of SCEst. While following the SLIC approach, this problem can not be solved satisfactorily. For example, if a `Suspend` statement is inside an `Abort` statement, the `Initialization` transformation could save this information at every `Suspend` statement by adding an annotation. Alternatively, before a `Suspend` statement is being transformed, the transformation could check if it holds an `Abort` statement in its body. If so, an annotation with extra information on the conditional, which should be placed after/before pauses, could be added to the `abort` statement. When the `Abort` statement is transformed in a following step, it could add the conditional after/before a generated pause. But this pattern would mix the different transformations and as mentioned, this would go against the SLIC approach because each feature should be transformed just once. Just by following the SLIC approach, this problem arose. However, it can be solved with an other structural incremental compilation approach. It starts its transformation at the innermost statement. After this single statement is being transformed, the intermediate model is saved to be inspected by the user after the whole transformation. The next enclosing statement will be transformed next and afterwards, again, the intermediate model is saved. Following that pattern, inner statements are being transformed first and therefore the problem of nested `aborts` and `suspends` does not exist. This approach is being explained in depth in the next chapter for future work (Section 7.2.4).

```
module SuspendAbort
{
  input bool A = false;
  input bool B = false;
  {
    bool abort1 = false;
    int v1 = 0;
    fork
      { // thread 1
        l1:
        if abort1 {
          goto l3;
        };
        l2:
        pause;
        if A {
          goto l2;
        };
        if B {
          v1 = v1 + 1;
        };
        goto l1;
        l3:
      }
    par
      { // thread 2
        l4:
        pause;
        if A {
          goto l4;
        };
        if v1 >= 2 {
          abort1 = true;
          goto l5;
        };
        l5:
      }
    join;
    l6:
  }
}
```

Listing 6.14. An `Abort` statement is inside a `Suspend` statement and the `Abort` transformation is executed before the `Suspend` transformation.

```
module SuspendAbort
{
  input bool A = false;
  input bool B = false;
  {
    bool abort1 = false;
    int v1 = 0;
    fork
      { // thread 1
        l1:
        if abort1 {
          goto l2;
        };
        pause;
        if B {
          v1 = v1 + 1;
        };
        goto l1;
        l2:
      }
    par
      { // thread 2
        l3:
        pause;
        if A {
          goto l3;
        };
        if v1 >= 2 {
          abort1 = true;
          goto l4;
        };
        l4:
      }
    join;
    l5:
  }
}
```

Listing 6.15. An `Abort` statement is inside a `Suspend` statement and the `Suspend` transformation is executed before the `Abort` transformation.

```
// [...]
  {
    bool abort1 = false;
    {
      int v1 = 0;
      bool f1 = false;
      fork
        { // thread 1
          l1:
          if f1 {
            goto l2;
          };
          pause;
          if A {
            abort1 = true;
            goto l2;
          };
          if B {
            if v1 == 2 {
              v1 = 0;
            };
            v1 = v1 + 1;
          };
          goto l1;
          l2:
        }
      par
        { // thread 2
          l3:
          pause;
          if A {
            abort1 = true;
            goto l4;
          };
          if v1 >= 2 {
            goto l3;
          };
          f1 = true;
          l4:
        }
      join;
      if abort1 {
        goto l5;
      };
    }
    l5:
  }
}
```

Listing 6.16. A `Suspend` statement is inside an `Abort` statement and the `Suspend` transformation is executed before the `Abort` transformation.

```
module AbortSuspend
{
  input bool A = false;
  input bool B = false;
  {
    bool abort1 = false;
    {
      int v1 = 0;
      bool f1 = false;
      fork
        { // thread 1
          l1:
          if f1 {
            goto l2;
          };
          pause;
          if B {
            if v1 == 2 {
              v1 = 0;
            };
            v1 = v1 + 1;
          };
          goto l1;
          l2:
        }
      par
        { // thread 2
          l3:
          pause;
          if A {
            abort1 = true;
            goto l5;
          };
          if v1 >= 2 {
            goto l3;
          };
          f1 = true;
          l4:
        }
      join;
    }
    l5:
  }
}
```

Listing 6.17. A `Suspend` statement is inside an `Abort` statement and the `Abort` transformation is executed before the `Suspend` transformation.

Chapter 7

Conclusion and Future Work

In this chapter a summary of the incremental compilation of SCEst and the evaluation of the implementation of former is given. Further, potential future work is proposed.

7.1 Conclusion

Within this thesis an incremental compilation of SCEst with the SLIC approach has been designed and implemented in KIELER. Therefore, already defined transformation rules were adapted and used. Also, an adjustment of the previously existing necessary meta-models as well as an additional meta-model for SCEst were needed for the successful design of the incremental compilation. While the step by step compilation of a SCEst program has in general been shown to be possible, two limitations arose.

First, when multiple `abort` and `suspend` statements are nested, the needed placement of conditionals before/after newly generated `pause` statements could not be done entirely. Therefore, a triggered `abort` or `suspend` might not be executed in its intended way.

Second, the intermediate model condition of the SLIC approach is not satisfied completely. While each intermediate model is a valid model, extra information on which signals are represented by which newly generated variables has to be saved in this implementation.

Four test cases showed that the number of generated SCL statements by the incremental compilation of this thesis was similar to the number of SCL statements generated by Rathlev's structural approach, as his transformation rules were reused during this incremental compilation. Additionally, with two test cases the produced SCL program had considerably less statements than the SCL program generated with Rathlev's approach. Since an interactive SLIC compilation approach was used, future implementations of optimizations can be integrated quite easily in the compilation chain.

However, the incremental compilation of SCEst allows for user to better understand the compilation as every intermediate model can be inspected.

7.2 Future Work

Before some new suggestions for future work are proposed, a few ideas for future work proposed by Rathlev [Rat15] are briefly mentioned, because they are still valid possible tasks. Note that this thesis, the incremental compilation of Esterel/SCEst, was one of his suggestions.

Besides types, functions and procedures, Esterel allows to import `tasks` from a host language. Since SCL and the following code generation do not include tasks, a transformation is not possible at the moment. Therefore, the code generation could be extended by a representing feature.

As mentioned in Section 1.1, the `weak suspend` statement of Esterel v7 can not be transformed to Esterel v5 kernel statements. Therefore, it is not included in this incremental compilation of SCEst. An inclusion would be nice to have since it adds expressiveness to Esterel, but it is by far not trivial as explained in detail by Rathlev [Rat15].

Lastly, a transformation from SCL to Esterel with the SLIC approach could be implemented in KIELER. Rathlev proposed a separation into programs which are B-constructive on the one hand and into programs which are not B-constructive but valid under the SC MoC on the other hand [Rat15]. The latter, which was approached by Schulz-Rosengarten [Sch16], is a non-trivial task, since sequentially constructive behavior would need to be expressed in a valid Esterel program.

7.2.1 Alternative Transformations using Kernel Translations

As described in Section 1.4, while using the SLIC approach, a source programming language is split into features. Each feature itself can have an arbitrary number of alternative transformations but at least one. As mentioned in Section 1.2.2, the implemented transformations of derived Esterel statements are written as direct transformations to SCL statements. For all derived Esterel statements an additional transformation could be added, which transforms the derived statement to Esterel kernel statements first. It would have to have a `produce` dependency, indicating which kernel statements might be created. This addition would provide alternative transformations for derived features for the user to choose from. If a user did not understand a specific direct transformation, the alternative could help in understanding the transformation of the statement and would therefore lead to a better understanding of the language itself.

This approach was not used during this thesis since more transformation steps would lead to an even more complex target code because it yields up the optimizations used in the direct transformations.

7.2.2 Transformation of Signals and referencing Statements

As mentioned in Chapter 6, the implementation of the incremental SCEst compilation created during this thesis does not satisfy the intermediate model condition of SLIC completely. This comes from extra information which has to be saved when signals are being transformed.

A different approach would be to transform all signals to new variables, but during the transformation, all statements that reference a transformed signal are being transformed as well. For some statements like the `sustain`, `emit`, `unemit`, `set` and `present` statement this would mean a complete transformation to SCL statements. For others, which reference a signal in an expression, the `ValuedObjectReference` would need to be changed to referencing the corresponding, newly created variable. This is the same approach used for SCCharts, as described in Section 3.1. However, as explained in Section 4.2.2, this would lead to a coarser partition of single transformations. Nevertheless, it would circumvent the problem of having to save a mapping from the old signals to the newly generated variables as extra information, as mentioned in Chapter 6.

During this approach, the set of features for the SLIC approach has to be adjusted. For example, the statements `sustain`, `emit`, `unemit`, `set` and `present` would be a single SLIC feature, e.g., a `signal` feature.

This could be implemented as an alternative compilation chain. Therefore, the user could decide which approach should be used. Again, this would give the user the freedom of choice and a possibility to compare the results of the different approaches and to choose the best result suited for a specific task.

7.2.3 Grammar Improvement

The implemented Xtext grammars for Esterel and SCEst needed to be improved regarding parser performance. As stated in Section 6.2, the auto-generated parser was not able to successfully parse deep nested statements in combination with Esterel's `EsterelParallel` statement. This problem occurred because the parallel statement does not start with a keyword, which resulted in the parser not being able to recognize a parallel statement from the beginning. Only when the `||` keyword was reached, it realized that all previous statements should have been in a thread. This problem made for future work as the possibility of deep nested statements is desirable. However, as mentioned in Section 6.2, A. Schulz-Rosengarten already modified the meta-model and the Xtext grammar of Esterel as part of the integration of this thesis' work into the KIELER project. Therefore, the implemented transformations of this thesis must be adjusted to these modifications in the future.

7.2.4 Structural Incremental Compilation with a New Compiler

During this thesis the KIELER compiler version 3 (codename *KiCool*) has been developed. The new compiler provides a more general configuration mechanism which does not depend on produce or not-handled-by constraints. Extending the compile framework could result in a structural compilation approach. The compiler is given a transformation rule for every SCEst statement, but the compiler dynamically constructs the compilation chain while compiling. This is done by inspecting the innermost statement (at the bottom of the syntax tree) first. A so called intermediate processor chooses the fitting transformation for this statement. After the statement has been transformed, the intermediate processor is activated again and inspects the next enclosing statement and chooses the proper transformation. Following this pattern the transformation stops when the root object is reached. With this approach a single transformation can be initiated multiple times. Following the whole transformation, every single intermediate model, resulting after each single transformation, can be inspected by the user since it has been saved.

This approach is not a single-pass approach as SLIC, because a single transformation can be activated more than once, as mentioned before. Additionally, this approach is finer than the approach of this thesis, since it allows the user to inspect transformations of single instances of statements.

A possible transformation of an example program is shown in Figure 7.1. The transformation starts at the innermost statement, the `Pause`. Note that the first concurrent thread in the source program is selected first and therefore the `pause` statement and not the `halt` statement is the first statement to be transformed. The next enclosing statement is the upper inner `Block` statement, which is therefore transformed at this point. After that, the `Parallel` statement is the next enclosing statement, but not all parallel threads have been transformed yet. Thus the transformation continues at the innermost statement of the second thread, the `halt` statement, which leads to the intermediate model `structural3`. The `halt` statement's enclosing statement is the lower inner `Block` statement, which is therefore transformed. Because all parallel threads have been transformed, the `Parallel` statement is transformed next, leading to `structural5`. Since the program is not fully transformed at this point, other transformations are done on the model, but they are not further shown for this example. Note that every listing displays a valid, self-contained intermediate model, which can be inspected by the user after the whole transformation is done. As shown, a transformation can be initiated multiple times (the `Block` statement in this example). This is additionally illustrated in Figure 7.2. It shows the new compiler view for this example.

This structural approach would eliminate the problem of nested `aborts` and `suspends`, shown in Chapter 6, as it was not a problem in Rathlev's approach, since the transformation starts at the innermost statement and goes from there step by step to the top of the syntax tree.

```
module structural1:
[
  [
   pause;
  ];
 ||
  [
   halt;
  ];
];
end module
```

```
module structural2:
[
  {
   pause;
  }
 ||
  [
   halt;
  ];
];
end module
```

```
module structural3:
[
  {
   pause;
  }
 ||
  [
   l1:
   pause;
   goto l1;
  ];
];
end module
```

```
module structural4:
[
  {
   pause;
  }
 ||
  {
   l1:
   pause;
   goto l1;
  }
];
end module
```

```
module structural5:
[
 fork
  {
   pause;
  }
 par
  {
   l1:
   pause;
   goto l1;
  }
 join;
];
end module
```

Figure 7.1. Transformation up to the `Parallel` statement with the new compiler.

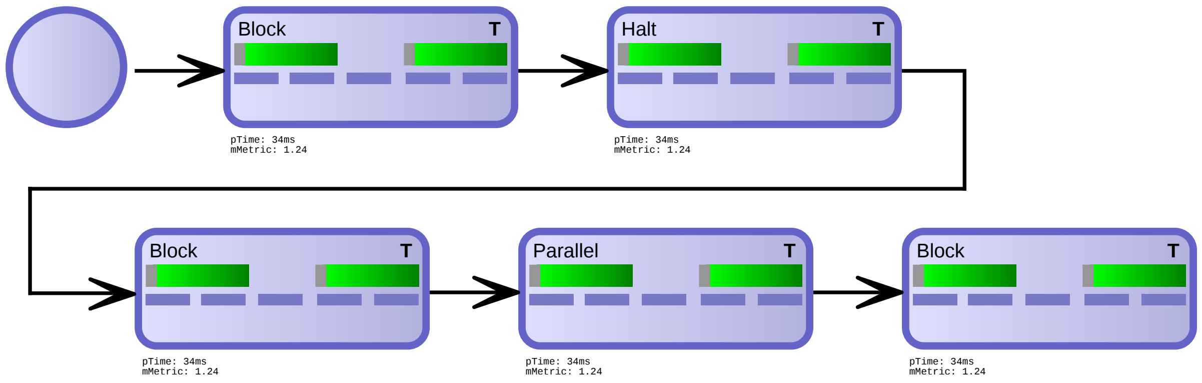


Figure 7.2. Example illustration of the new Compiler View for `structural1` of Figure 7.1.

Bibliography

[And03] Charles André. *Semantics of SyncCharts*. Tech. rep. ISRN I3S/RR–2003–24–FR. Sophia-Antipolis, France: I3S Laboratory, Apr. 2003.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. "The Synchronous Languages Twelve Years Later". In: *Proc. IEEE, Special Issue on Embedded Systems*. Vol. 91. Piscataway, NJ, USA: IEEE, Jan. 2003, pp. 64–83.

[Ber00] Gérard Berry. *The Esterel v5 language primer, version v5_91*. ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf. Centre de Mathématiques Appliquées Ecole des Mines and INRIA. 06565 Sophia-Antipolis, 2000.

[Ber02] Gérard Berry. *The constructive semantics of pure Esterel*. Centre de Mathématiques Apliqées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France: Draft Book, Version 3.0, Dec. 2002.

[Ber91] Gérard Berry. *A hardware implementation of pure ESTEREL*. Tech. rep. de recherche 1479. INRIA, 1991.

[Ber93] Gérard Berry. "The semantics of pure Esterel". In: *Proceedings of the Marktoberdorf Intl. Summer School on Program Design Calculi*. LNCS. http://citeseer.ist.psu.edu/berry93semantics.html. Springer-Verlag, 1993.

[BLL+05] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. *Heterogeneous concurrent modeling and design in Java, volume 1: introduction to Ptolemy II*. Tech. rep. EECS Department, University of California, Berkeley, July 2005.

[Bol03] Azad Bolour. "Notes on the Eclipse plug-in architecture". In: *Eclipse Corner Articles* (July 2003). www.eclipse.org/articles.

[Bt00] Gérard Berry and the Esterel Team. *The Esterel v5_91 system manual*. http://www-sop.inria.fr/esterel.org/. INRIA, June 2000.

[EZ07] Stephen A. Edwards and Jia Zeng. "Code generation in the Columbia Esterel Compiler". In: *EURASIP Journal on Embedded Systems* Article ID 52651, 31 pages (2007).

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Edinburgh, UK: ACM, June 2014.

[HMA+13] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. *Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation*. Technical Report 1308. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2013.

[Lee06] Edward A. Lee. "The problem with threads". In: *IEEE Computer* 39.5 (2006), pp. 33–42.

[MG06] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (2006). Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), pp. 125–142. ISSN: 1571-0661. DOI: DOI:10.1016/j.entcs.2005.10.021. URL: http://www.sciencedirect.com/science/article/B75H1-4JFR8K3-B/2/cd561440d16d44082d7b63da61c70252.

[Mot17] Christian Motika. *Sccharts—language and interactive incremental implementation*. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2017. ISBN: not assigned yet.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. New York, NY, USA: Springer, 1995. ISBN: 0-387-94459-1.

[PEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

[Pot02] Dumitru Potop-Butucaru. "Optimizations for faster simulation of Esterel programs". PhD thesis. Ecole des Mines de Paris, France, Nov. 2002.

[Pow99] James Power. "Notes on formal language theory and parsing". In: Maynooth, Co. Kildare, Ireland, 1999.

[PTH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. "Synthesizing Safe State Machines from Esterel". In: *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '06)*. Ottawa, Canada, June 2006.

[Rat15] Karsten Rathlev. "From Esterel to SCL". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/krat-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Mar. 2015.

[RB06] Jim des Rivieres and Wayne Beaton. "Eclipse platform technical overview". In: *Eclipse Corner Articles* (Apr. 2006). www.eclipse.org/articles.

[RSM+15] Karsten Rathlev, Steven Smyth, Christian Motika, Reinhard von Hanxleden, and Michael Mendler. "SCEst: Sequentially Constructive Esterel". In: *Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '15)*. Austin, TX, USA, Sept. 2015.

[Rüe11] Ulf Rüegg. "Interactive transformations for visual models". Bachelor Thesis. Kiel University, Department of Computer Science, Mar. 2011.

[Sch16] Alexander Schulz-Rosengarten. "Strict sequential constructiveness". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Sept. 2016.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – Putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[Tar05] Olivier Tardieu. "A deterministic logical semantics for Esterel". In: *Electronic Notes in Theoretical Computer Science* 128.1 (2005), pp. 103–122.